TACHYON: CUSTOMIZABLE PROGRAM ANALYSIS

VIA

GENERIC ABSTRACT INTERPRETATION

by

Nathaniel D. Osgood

B.S., Electrical Engineering and Computer Science
Massachusetts Institute of Technology, 1990

S.M., Electrical Engineering and Computer Science
Massachusetts Institute of Technology, 1993

SUBMITTED TO

THE DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

AT THE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

FEBRUARY 1999

Signature of Author.....
Department of Electrical Engineering and Computer Science
January 29, 1999

Certified by.....
Stephen A. Ward
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by.....
Arthur C. Smith
Chairman, Committee on Graduate Students

TACHYON: Customizable Program Analysis

via

Generic Abstract Interpretation

by

Nathaniel D. Osgood

Submitted to the Department of Electrical Engineering and Computer Science
on January 29, 1999 in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy in
Computer Science

ABSTRACT

This thesis formulates and implements a new approach to program analysis based on the principle that analyses of a program's behavior can be implemented as executions of the program under an abstract semantics. We have used this principle to create a generic analysis framework that can be parameterized at analysis-time to collect user-specified information on program behavior.

The first step in this approach is the characterization of generic abstract execution in a way that segregates the abstract domains that approximate run-time states from those that collect user-defined information on the execution history of the program. Secondly, we characterize abstract approximations to values and states as partitioned into "semantic domains" that adhere to certain interfaces. Each such semantic domain encapsulates both the information approximated or collected by an abstract program quantity and the semantic rules for manipulating that information when different operations are encountered during analysis. Finally, we describe and implement a generic analysis engine that allows the user to "plug in" any user-defined domains that adhere to the required interfaces. The execution of this engine conducts abstract interpretation of a program using the specified domains. By virtue of having segregated the domains necessary for approximation and collection of information, the user can independently vary the domains that determine the type of information to be collected and those that dictate the precision with which the analysis is to be conducted.

To boost the speed of the generic analysis, I have implemented a strategy to compile source language programs into a "compiled analysis" executable whose function is to perform customizable abstract execution on the original source program. Thus, while past techniques have taken the traditional approach of hard-coding analyses to collect a fixed set of information on any program, this strategy collects a variable set of information on a fixed program.

A prototype system has been implemented to test and refine these ideas, including two compilers to the abstract semantics and a number of example semantic domains. While its performance is compromised by the use of a general abstract interpretation algorithm, the system permits easy creation of and modification of custom analyses.

Thesis Supervisor: Stephen A. Ward
Title: Professor of Electrical Engineering and Computer Science

TABLE OF CONTENTS

Number *Page*

LIST OF FIGURES

Number *Page*

LIST OF TABLES

Number *Page*

ACKNOWLEDGMENTS

My graduate school trajectory has been very unusual one, punctuated as it has been by a multi-year hiatus. It has also been a time over which my interests have evolved a great deal. I'd like to offer thanks to all who have cushioned the many transitions that have been involved. I am deeply grateful for having such a wonderful family, friends, and an extraordinarily understanding advisor. The gratitude I feel to all is tremendous, and I am at a loss to express it here. I am particularly grateful to my fiancée XQ, for her deep patience, encouragement and support. She has made these years especially precious and memorable. I am also exceptionally thankful for my close friendship with Shaun Kaneshiro, who has been an enormous help and support throughout the years. Shaun has lent me a hand in so many tricky situations, and has been a tremendously valuable spiritual companion as well. The MIT Buddhist Association has offered deep support throughout my time as a doctoral student. Were it not for my contact with BAMIT, this thesis would never have been written. Hong-Tat Ewe warrants particularly mention in this regard, and for his patience and thoughtfulness I remain deeply grateful. I am also thankful to Benny Budiman for his companionship, wisdom and hospitality. My friendship with May Ku has brought me great happiness, and served as a joy through many rough times. Hank Taylor has been a truly exceptional friend in every respect, a dynamic but mindful business partner, and a tireless mentor. It is a great honor and joy to spend time with someone who has so thoroughly eschewed established conventions of what is deemed "worthy" and embraced life so consciously and fully. Our time together has enriched my life in many ways. I look forward to many "toilet hacking" escapades with Hank in the future. Ellen Spertus has been a giant help and a kind and thoughtful companion through the years. Her generosity and tireless support are deeply appreciated. Chris Tsien has also been a long-standing source of comfort and fun, and to her I owe a great deal.

It is difficult to imagine having made it through graduate school without the friendship of other key individuals, such as Mark Smith, Lily Lee, and Ye Xu. They have been very important in many ways, and I am grateful for their companionship. During the first part of graduate school, I had the good fortune to enjoy a tremendously enjoyable and understanding set of apartmentmates, and I am privileged to call them friends. Yanwu Zhang, Sramana Mitra, and Yanqing Du have each added a great deal of joy and fun to my life.

I'd also like to offer strong thanks to the staff of Woburn Dental Associates (particularly Patti Buckley and Dr. Agular), who have been exceptionally generous with their time and care, and have offered the warmest of friendship. My times in the dental office rank among the most enjoyable points in my life, and I look forward to future visits. I am very thankful for having met such a wonderful group of people.

Anne McCarthy also deserves particular mention for her continuous formal and informal help throughout all of my convoluted graduate school career. Her attention has been a tremendous aid and comfort in some tricky situations. In a world where the dedicated secretaries frequently work the true magic, she stands out as a true sorcerer. I have much to learn from her on the art of life, and the life of art.

Thanks so much to Professors Gerry Sussman and Tom Knight for serving on the thesis committee. Their comments helped spin gold from hay, and their guidance is among the most valuable I have received in graduate school. I am very grateful to them for their time and patience.

Finally, I would like to offer heartfelt thanks to my family for their help and support throughout my entire education. I could not have asked for a more supportive, warmer, and more understanding family. My parents have been particularly generous and helpful during my graduate school career, and I could not

have completed my degree without their support and love.

I have enjoyed the enormously privilege of having my grandmother as my primary conversational companion throughout the period in which I worked on this thesis. Her spirit, resolve, tolerance, and understanding have been a true inspiration to me, and I have enjoyed my time with her immensely. In a symbolic sense, she has been the fourth – and most important – member of the thesis committee, and the member who has taught me the most of enduring value.

Sincere thanks to all!

Chapter 1 Introduction

1.1 Overview

This thesis formulates a methodology that substantially lowers the effort and expertise required to create customized analyses of program behavior. This chapter briefly examines the motivations for this approach and provides a glimpse of both the techniques that support it and of how the approach would be used. We then examine the application of these techniques to a toy example, and close with an overview of the thesis organization.

1.2 Motivations

1.2.1 Software Trends

The past decade has seen a striking growth in the use and diversity of software. Programs span not only the full range of conventional programming languages, but are increasingly written for embedded scripting engines (e.g. in applications and browsers), as declarative statements in spreadsheets, mathematical modeling tools, and constraint-based systems. Far from being an activity relegated only to software labs and IT shops, programming is increasingly commonplace within general-purpose office environments, and is frequently conducted by individuals with little or no formal training in software design or programming practice. At the same time, software companies (and, to a lesser degree, larger corporations) continue to undertake large-scale software projects with a breadth and aggregate complexity that dwarfs previous systems.

Whether they are computer novices modifying scripts for their home page or software engineers working on a large-scale integrated system, programmers are increasingly faced with the need for more effective tools to understand the operation of their programs.

Sensing this widespread need, software toolmakers have responded with a variety of program analysis packages. Such systems (typically aimed at professional developers) provide the software developer with a variety of insights into program operation: Instrumented testing tools attempt to catch subtle memory or pointer errors or API conformance bugs at runtime, testing systems that analyze system coverage, profiling systems aim to give a developer better insight into software system performance, and debugging packages compete to provide better information concerning program behavior prior to a breakpoint or error. Other less widely used systems offer information on the time behavior of a coupled system of

ordinary differential equations, check test vector coverage or Y2K compliance, and allow for visualization of program operation and heap contents.

1.2.2 Shortcomings of Today's Systems

Unfortunately, the analysis marketplace is artificially fragmented. The independent, monolithic interfaces sported by analysis tools seem anachronistic indeed at a time when componentization has transformed almost all categories of software. This is particularly regrettable in light of the fact that while they provide access to very different kinds of information on program behavior, most of such systems conduct analysis using fundamentally similar techniques. A cheaper and more natural solution would mirror the underlying similarities uniting program analyses by providing individual bundles of analysis rules as componentized extensions to a core analysis implementation.

Another – and more pressing –problem with existing tools lies in the fact that they are typically inflexible in terms of the type of information collected on program behavior and the rules by which that information is collected. Existing analysis packages offer the user little capacity for the user to customize, refine or change the precision of the analysis rules. In the best case, the need to perform a new type of analysis requires the purchase of a new software package rather than building on top of existing packages. For cases in which no existing package supplies the sort of information that is needed, the user is faced with the option of doing without or crafting their own analysis system – a challenge that can be taken up by only a small fraction of software professionals.

A final shortcoming of existing analysis systems is that they are artificially divided into two extremes:

- Run-time analysis systems require complete specification of all aspects of the program execution context, and can run only on a single input at a time. The analysis conducted by such systems is precise, but its termination and running time are contingent upon those of the program being analyzed. Examples of popular run-time analyses include systems that check for API conformance, memory leaks, loose pointers, collect profiling or code coverage statistics, etc.

- Compile-time systems analyze a program without allowing *any* specification of knowledge regarding that context. Analyses that collect information for program optimization, program slicing, or for use in debugging frequently fall into this category. Such systems guarantee timely termination of analysis, but cannot take advantage of any knowledge of program input.

While such categorization will sometimes fit the needs of a user performing an analysis, more typically the user will wish to conduct the analysis given certain *partial* information regarding the program execution context. The "binary epistemology" adhered to by most analysis systems seems increasingly anachronistic in a world where user interaction and the acquisition of data manipulated by programs frequently takes place in an asynchronous manner. (For example, while the data acquired from a database may not be known until the program is run, the database schema may be considered fixed at the time the program is written.) While the availability of information at analysis time concerning program execution context can substantially improve the quality of the analysis results, the vast majority of existing analysis systems lack the ability to take advantage of such information.

1.3 A New Methodology

This thesis introduces a new program analysis methodology that attempts to ameliorate the pitfalls discussed above. There are four key components to this approach:

- **Semantic Parameterization in a Generic Analysis Framework**. The first step in the methodology is the formulation of a common mathematical and implementation framework for performing analyses of program behavior. Within this thesis, we take advantage of the fact that although different analyses may differ in the types of data collected and the semantic rules used to create that information, a broad class of analyses can be formally characterized as the results of *abstract executions* of a program – runs of the program in which the states and values being manipulated are *approximations to* the states and values present in run-time program operation (or to their histories). [Cousot and Cousot 1977] This formalization provides a precise criteria for establishing the correctness of an analysis, and captures the inherent commonalties shared by seemingly disparate analyses of program behavior. The critical observation that makes this thesis possible is that the common mathematical underpinning of analysis algorithms allows for a common *implementation* as well: Many analyses can be expressed as simple polymorphic parameterizations of a generic analysis algorithm by "semantic domains" components whose semantic rules for different value or state operators capture the particular information of interest to the user. This thesis specifies a set of relatively compact, language-independent interfaces for these semantic domains. Implementation of these interfaces allows a domain component to be dynamically "plugged into" a unified implementation framework capable of executing an unbounded set of analysis algorithms on a particular program. This approach is mirrored by a change in perspective on the relationship between analysis and program. Rather than taking the traditional "procedural" perspective of analysis in which a program is viewed as a data object to be passed to one or more analysis routines, this thesis

adopts an "object oriented" view of programs in which the program (or, more precisely, its derivative analysis object) maintains interfaces for analyzing itself, and where the *analyses* are passed to this program as objects.

- **Capacity to Independently Vary the Information to be Collected and the Precision with which to Collect It.** Program analyses have traditionally been implemented so as to collected information on program behavior only in the presence of complete information concerning program values (and input), or with some specific representation of the possible values taken on by program quantities (e.g. a predicate indicating that the quantity is known to lie within some interval or in a set of possible values). This linkage between analysis precision and the rules for collecting the information allows for the formulation of efficient algorithms, but prevents the same information from being collected at different points along the precision/analysis run-time curve, and in the presence of different amounts of information concerning the program inputs. Taking advantage of the structure of the information flow during abstract interpretation, TACHYON separates the "semantic domains" by which analyses are parameterized into two categories: Domains whose function is to *approximate* run-time quantities in the current state for the purposes of more tightly bounding the set of concrete states that are possible at run-time, and domains which *collect* information on the computation history of a program. By allowing these domains to be specified separately, the user can make use of many different types of approximating domains (and levels of knowledge about program input) while still collecting the same type of information on program behavior. Each type of approximating domain will imply different analysis running times and levels of accuracy of the analysis results. Conversely, analysis creation process simplified by the ability to reuse a particular approximating domain with a wide variety of collected domains.

- **Scalable Approximating Domains.** While the capacity to interchangeably make use of different approximating domains provides the user with a rather coarse-grained means of adjusting analysis precision, portions of this work examined finer-grained approaches for trading off analysis precision and efficiency. In particular, the research described in this thesis focused on the opportunities for moving beyond the assumptions adhered to by most analysis systems, in which program values are regarded either as completely known (and associated with particular concrete values) or completely unknown. Guided by both a clear model of uncertainty in program analysis and by the shortcomings of past approaches to stronger representation, we have formulated two "heavy-weight" value abstractions that can serve as "approximating domains" within the analysis. These domains allow for graceful degradation of the precision of value approximation in the presence of limitations on running

time or on the level of knowledge concerning the program input. One of the value abstraction we employ is distinguished from past representations in that it is based on a formal model of programmatic uncertainty within a program. This abstraction contains a (potentially incomplete) set of samples extracted from a value's *sample space*, where each element of that value is associated with a possible execution context for the program. In effect, this allows for simultaneously running the program on some user-determined set of possible inputs.

- **Technique for Performing Compiled Analyses.** The methodology for performing generic, extensible analysis described above provides the user with a simple and convenient means of creating, customizing and running analyses of program behavior, but this approach does not come without costs. Most importantly, generic analysis is achieved by means of ubiquitous polymorphism, which carries with it a considerable performance burden. Most previous analysis systems have achieved high performance through specialization – namely, by customizing the analysis algorithm to the particular analysis information that needs to be collected. The specialized algorithm is then run over the intermediate representation of whatever program is to be analyzed. This thesis takes an orthogonal approach to the problem: Rather than specializing the analysis algorithm to the information to be collected and running that algorithm on *any* program, TACHYON specializes the analysis algorithm to the *particular program to be analyzed*, and then allows that analysis to run with a wide range of user-defined semantics domains. Figure 1 schematically illustrates this approach in more detail. Whenever the user compiles a module, the system specializes an abstract interpreter to that module in order to generate a very special type of object module. The job of this object module is to perform analysis of the *original* source code module by means of abstract execution using the extensible abstract semantics described above. When this object module is linked with other similar "compiled analysis" object modules and run, it will serve as part of a global analysis of program behavior. While the costs associated with extensible analysis are still heavy, the technique of "compiled analysis" allows for higher performance implementations of user-defined analyses.

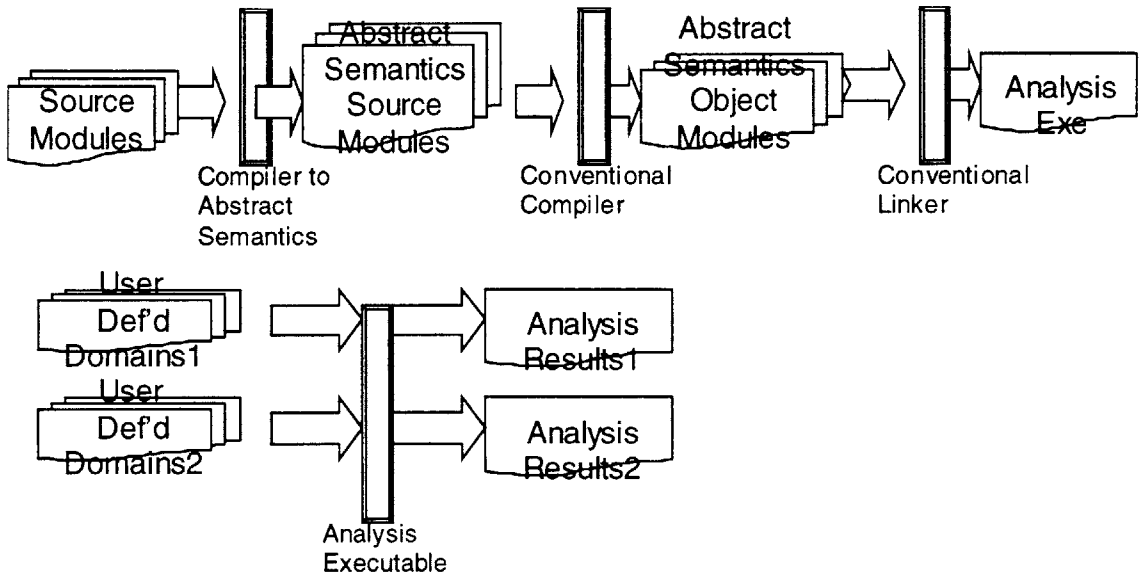

Figure 1: The "Compiled Analysis" Approach.

1.4 A Concrete Example

1.4.1 Introduction

The previous section surveyed important themes in TACHYON's approach. This section provides a concrete example that offers a glimpse of how the different parts of the existing analysis system work together. In order to avoid the danger of getting lost in code, the example itself is small and simplistic. The same approach, however, scales directly to programs of much greater size and complexity.

We will examine a hypothetical situation in which a user employs the generic analysis methodology to gain insight into the behavior of a program. While this thesis devotes most of its discussion to the analysis of programs written in conventional programming languages (such as the toy language presented in Section 4.4), the approach taken is a general one, and can be applied to any program associated with a well-defined semantics. To underscore the generality of the method, we have chosen as the subject of the example a piece of numeric code. The use of a semantically parameterizable generic abstract execution framework allows the user great flexibility in choosing different componentized "lenses" through which to view and understand the program's behavior. The generic analysis framework also permits the same sort of information to be collected using many different approximations to numeric values, with varying consequences as to the running time and accuracy of the analysis results.

1.4.2 The Program

The mathematical code we will analyze implements a mathematical model that was constructed using conventional system dynamics modeling methodology. [Forrester 1968] The model is formulated to capture in an aggregated way some of the links between corporate productivity, morale, and external demand, and to provide insights into the implications of these relationships for the response of the system given different externally experienced *demand*. Figure 2 depicts the model using a "stock and flow" visual representation common in that field. Each variable in this model is associated with precisely defined equations, which can be textually rendered. A fragment of the source code associated with this program is given in Figure 3.

The fact that the model under study is decidedly non-linear makes the behavior of the program very difficult to predict from the source code. Analyses of program behavior can offer great help in understanding the dynamics of such a system. Traditional modeling frameworks have addressed this need by allowing the user to run the model on particular inputs, allowing for the creation of graphs of system variables over time, and possibly permitting the collection of predefined statistics on variable behavior. By virtue of providing a single, unified framework for conducting easily customizable analyses comparison, the parameterized generic analysis approach offers a particularly attractive alternative. Most importantly, it eliminates the dependency of the user on the software vendor to provide the precisely desired analysis tools. By "running" the system on different sorts of "semantic domains", the user can collect any variety of information on system behavior, and can approximate potential system input and functioning to whatever precision is desired. We will now turn to an examination of how a user might make use of these capabilities to gain better insight into model behavior.

1.4.3 Compilation

The first step towards analyzing the system of Figure 3 is the compilation of the system source code (such as that shown in Figure 3) into a "generic analysis executable." To create the generic analysis engine, an "abstract semantics" compiler (one of two described in Chapter 8) is used to convert the program shown in Figure 3 into analogous code in the abstract semantics. (In other words, the original program is compiled into code whose execution approximates and collects information on the execution of that original code). For example, the translation of a particular fragment of that code is shown in Figure 4.

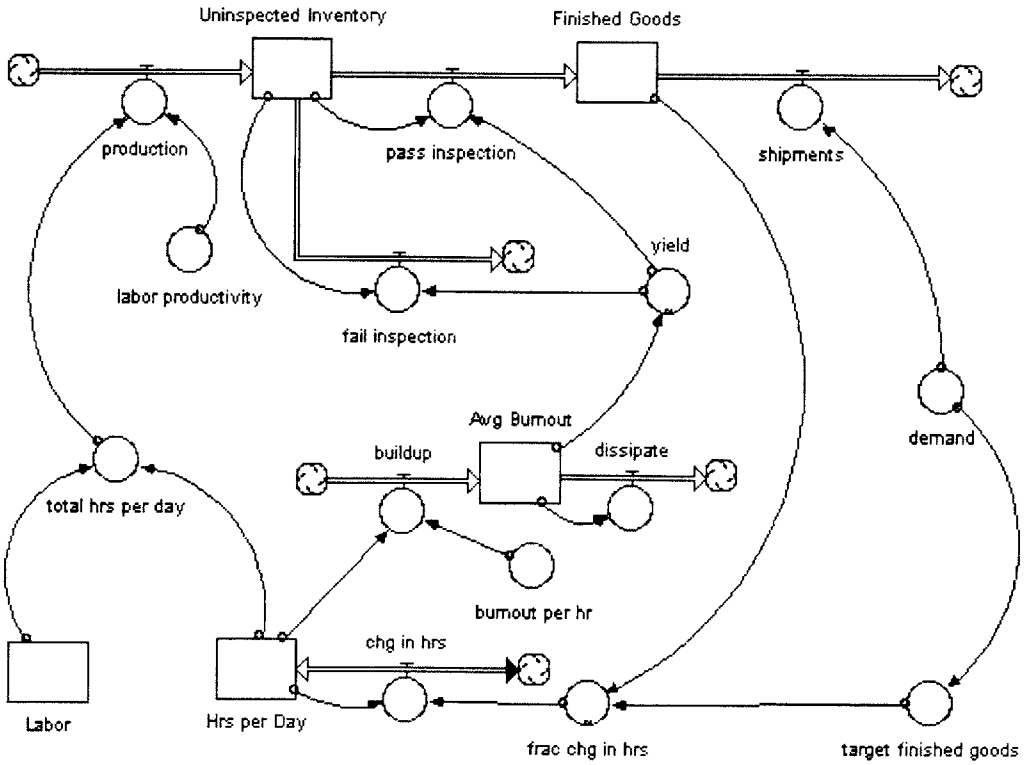

Figure 2: The Simple Numerical Model to be Analyzed.

```
Avg_Burnout(t) = Avg_Burnout(t - dt) + (buildup - dissipate) * dt
INIT Avg_Burnout = 0

buildup = IF Hrs_per_Day > 8 THEN ((Hrs_per_Day-8)*burnout_per_hr) ELSE 0
dissipate = Avg_Burnout*.1
Finished_Goods(t) = Finished_Goods(t - dt) + (pass_inspection - shipments) * dt
INIT Finished_Goods = target_finished_goods
pass_inspection = Uninspected_Inventory*(yield/100)
shipments = demand
Hrs_per_Day(t) = Hrs_per_Day(t - dt) + (chg_in_hrs) * dt
INIT Hrs_per_Day = 8
```

Figure 3: Example Fragment of Model Code (in IThink™ Syntax)

```
Finished_Goods(t) = Finished_Goods(t - dt) + (pass_inspection - shipments) * dt
```

$\Rightarrow$

```
Finished_Goods = statePrototype.CurrentStateCurrentInterface().operatorAssignmentToDoubleVariab
leFilter(7, Finished_Goods, Finished_Goods.operatorAdd(pass_inspection.operatorSub(shipments).o
peratorMul(AbstractDt)).operatorMap(zeroCutoff), 0);
```

Figure 4: A Stock Update Statement and its Analogue in the Abstract Semantics.

The "abstract semantics" code maintains a very precisely defined relationship to the code from which it was translated (an issue explored in greater depth in Chapter 2 and Chapter 3). While the operation of the

original (or "standard semantics") code evaluates expressions of double precision values, the translated code evaluates abstract analogues to those expressions on abstract double precision values (i.e. on values that approximate and collect information on the concrete values). Where a standard semantics simulation would update the program state by assigning a variable some calculated value, the abstract semantics implementation notifies and updates the *abstract* state. Because the user can choose the particular semantic domains to be associated with the abstract state and abstract value (e.g. the particular semantic rules to be used in the implementation of *operatorMul* and *operatorAssignmentToDoubleVariableFilter*), a particular sequence compiled abstract semantics code can realize a wide variety of analysis implementations.

Figure 5 shows the interface to the abstract semantics compiler, as implemented in the prototype system. The compiler translates IThink™ code specified in the top pane or in a file into its abstract semantics analogue (seen on the bottom pane, and implemented in Java). The button on the lower right compiles the translated code into the generic analysis engine (comprised of the compiled code, a user interface component, and a run-time library.) The next section discusses how this general-purpose analysis engine can be parameterized to collect a variety of types of information, and to approximate program values in different ways and to different precision.

1.4.4 Analyses

1.4.4.1 Execution with Standard Supporting Domains

With the creation of the generic analysis executable using the compiler shown in Figure 5, the user is ready to conduct analyses. By running the executable, the user loads the generic analysis engine, which can then be used to perform any sort of user-specified analysis.

Before the analysis begins, the user must select the particular type of information to be collected and the precision with which to approximate the values circulating at runtime. To allow this selection, upon startup the analysis executable presents the user with the dialogue box shown in Figure 6. Within this dialogue, the user is asked to select two types of "semantic domains" by which to parameterize the analysis:

- **Collected Domains.** "Collected" domains encapsulate the type of information to be collected by the analysis and the rules for deducing it. As suggested by Figure 6, the user can "mix and match" collected domains so as to simultaneously collect information on many different characteristics of program behavior. A given collected domain must be able to transparently scale its operation from cases in which the values being manipulated in the program are precisely known to cases in which

those values are completely unknown. This allows the same sort of information desired to be collected given any sort of constraints on running time or on the amount of information known regarding program inputs.

- **Approximating Domains.** These domains (which are also termed "guiding" domains), determine the means and precision by which run-time values and states are approximated during analysis. (For example, as detailed in Chapter 10 and Chapter 11, value quantities can be approximated as concrete values, as intervals, as sets of values in a sample space, as disjoint sets of intervals, etc.) Heavier-weight representations typically allow for more precise analysis, but entail longer running analysis times. By means of a convenient overloading of purpose, the approximating state domains are responsible for approximating the program state, but also can be used to enforce alternative patterns of program input in which the user is interested. As suggested by Figure 6, only a single approximating domain may operate in a given analysis for each type of object.

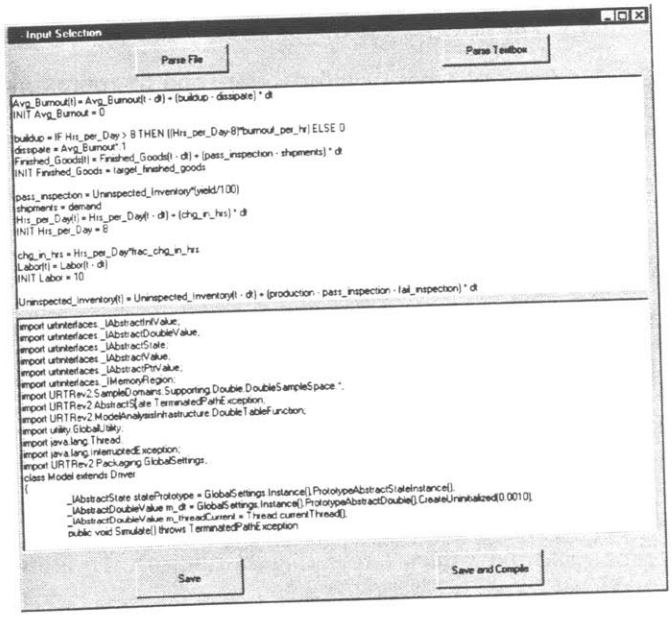

Figure 5: Compilation of a Standard Semantics Program into its Abstract Semantics Analogue.

Speaking informally, we can think of the collected domains as specifying the type of information to be collected, while supporting domains determine the precision with which to collect it. These two types domains are characterized more formally and in greater depth in Chapter 3.

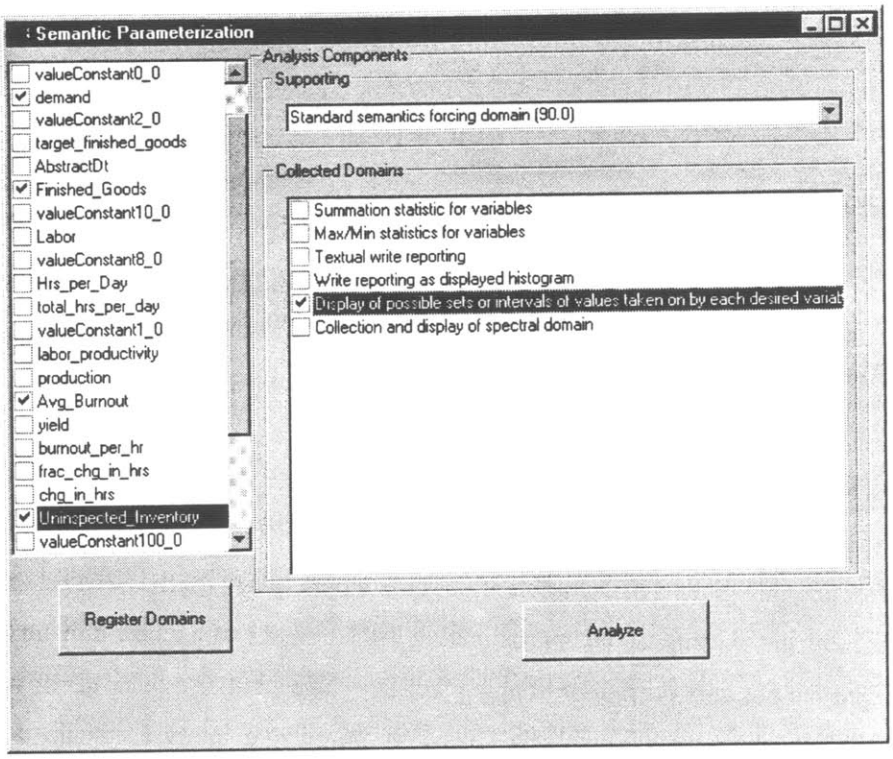

Figure 6: Selection of the Semantic Domains for Analysis.

For the particular example to be considered here, suppose that the user is interested in examining the system response's to a particular "step function" increase of the external demand input variable. To accomplish this, she makes use of two domains. As a collected domain, the user selects a domain whose semantic rules collect information on the possible values taken on by particular semantic variables over time, and display those profiles as graphs. In the presence of imprecision regarding the run-time value taken on by a variable, the domain collects information on and shows graphs of maximum and minimum possible values. Because the user here wishes to run the program on a particular, precisely specified input, the approximating domain is very simple: She can make use of abstract state and abstract value approximating domains that approximate values either as completely known, or completely unknown. One exception to this fact is that the abstract state domain chosen by the user makes use of the overloading mentioned earlier in order to allow the user to "force" a desired input variable (*demand*) to take on the specified user-defined value.

Running the model using these two approximating and collecting domains, the user simulates the response of the model to a particular value for the variable "demand". Outputs from such runs are shown in Figure 7. Perhaps surprisingly, the model exhibits instability. Her interest piqued, the user seeks further investigation as to the range of program response that is possible for different values of the input.

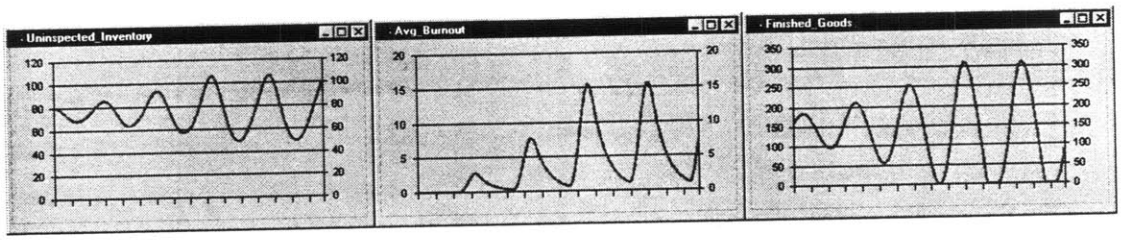

Figure 7: Operation of the Program on a Concrete Approximation to Program Variables, Using a Visualization Collected Domain.

1.4.4.2 Execution with Other Abstract Supporting Domains

1.4.4.2.1 Executing the Model on Interval Values

Having observed the general system response to a particular input, the user is interested in understanding the modes of behavior possible given a *distribution* of possible inputs values for the *demand* variable. To this end, the user replaces the existing trivial approximating value domain with a new domain that allows abstract values to represent not only *particular* double-precision values, but also *intervals* of such values. This "interval domain" (a close double-precision cousin of the integer interval domain described in Chapter 10) implements the semantic rules for each value operator according to the traditional rules for interval arithmetic (e.g. adding together two intervals requires simply adding their endpoints, etc.). Figure 8 shows the implementation of the multiplication operator for the interval semantics; other operators (such as division) are more difficult to accurately model.

```
private DoubleIntervalDomain operatorMul(DoubleIntervalDomain argRHS)
{
    double a = this.m_lower * argRHS.m_lower;
    double b = this.m_lower * argRHS.m_upper;
    double c = this.m_upper * argRHS.m_lower;
    double d = this.m_upper * argRHS.m_upper;
    return new DoubleIntervalDomain(min(a,b,c,d), max(a,b,c,d));
}
```

Figure 8: The Code to Implement the Multiplication Operator for the Interval Domain.

It is important to recall that the interfaces to all collected domains require that such domains be able to operate in the presence of arbitrary uncertainty regarding the values circulating in the program. As a result, the same collecting domain can be used to analyze execution of the program under the interval approximating domain as was used for the earlier run on concrete values.

As shown in Figure 9, running the program under the interval approximating domain and the value visualization collecting domain gives the user some insight into the ranges of program behavior that can be seen for a variety of inputs. Unfortunately, the rules of interval arithmetic used by the interval

approximating representation are frequently grossly over-conservative. This failing manifests itself in the context of value combination, and arises in large part from the inability of the interval representation to take into account *dependencies* between values. For example, the rules for interval multiplication specify that given value *a* lying in the interval [-1,1], *a/a* could lie anywhere along the real number line – an egregiously conservative approximation to the true interval [1,1] in which the result must lie.

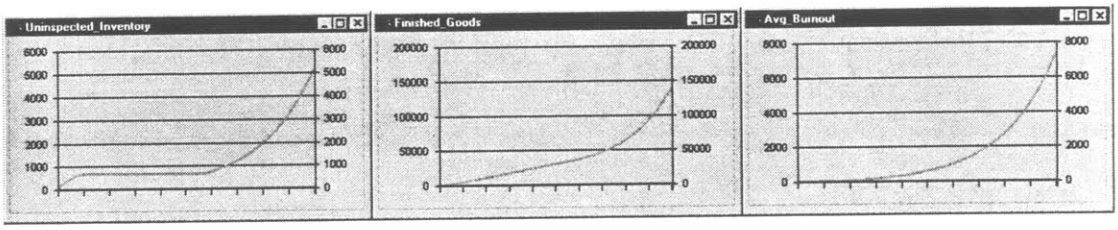

Figure 9: Estimates as to the Numeric Bounds of Program Variables, as Estimated with the Interval Approximating Representation.

One consequence of this imprecision is that the execution of the model on the interval value supporting domain yields an approximation to the behavior of the program that is sound but very crude. The vertical scale in Figure 9 documents the rapid divergence of the system on the range of inputs specified. As is documented in Appendix 1 and Appendix 2, this inaccuracy arising from value combination is typical for cases in which the values being combined exhibit high amounts of statistical dependence.

1.4.4.2.2 A Higher-Fidelity Supporting Domain

Still hoping to more accurately understand system behavior under a variety of inputs, the user discards the interval value representation, and makes use of an alternative approximating value domain – the "sample space" domain introduced in Chapter 11. This domain approximates each program value as a random variable defined over some (potentially multidimensional) user-defined sample space. Each configuration in the sample space is associated with a particular value for the value. Analysis of the program using such an approximating value domain in essence represents the superposition of the analysis of the program in each of a set of possible execution contexts.

In the particular case at hand, the user again makes use of the forcing approximating state domain to force the variable *demand* to take on a particular sample space value. To do so, the user specifies the set of values taken on by that variable for the different possible configurations in sample space.

While the user has exchanged an interval approximating representation for a sample space domain, she leaves the *collected* domain unchanged. The system will thus continue to collect and display information

on the value (or bounds) associated with each selected variable over time. The analysis is then carried out. The results of this analysis are depicted in Figure 10. In contrast to the analysis conducted using the interval domain, the system maintains tight bounds over the system's behavior and the range of variation of different program values.

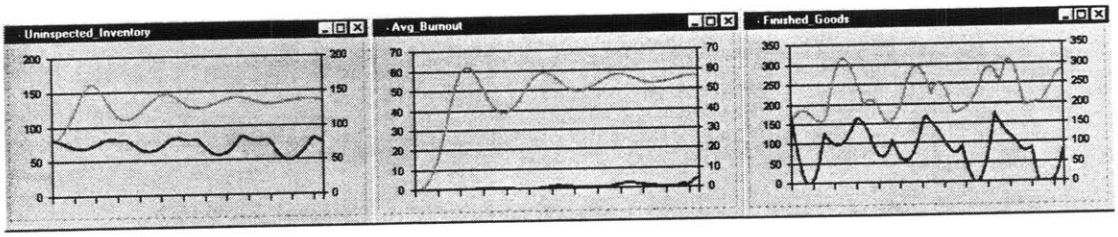

Figure 10: Effective Bounds on Ranges Associated with Model Variables attained through Abstract Execution with the Sample Space Approximating Domain.

Viewing the envelope associated with program behavior over time, the user notices that while the responses to *all* possible values for demand are tightly clustered together for during the initial stages of model evolution, they appear to go out of synchrony in later stages. This leads to a substantially broader envelope in time periods beyond those shown in Figure 10.

In order to explore this phenomenon, the user turns to a new visualization-oriented collecting domain. In particular, the user employs a collecting domain that allows for visualization of values approximated by sample space value approximating domains by depicting the values associated with program quantities for *each* of the different elements of the sample space. Figure 11 shows the results generated by this domain for the particular variables displayed earlier. As before, the horizontal axis of the graph is time, while the vertical axis corresponds to the value taken on by the variable. Each curve within the graph is associated with a particular element of the sample space (i.e. a particular input for variable *demand).* The graph thus displays the values taken on by a particular variable for each configuration over time.

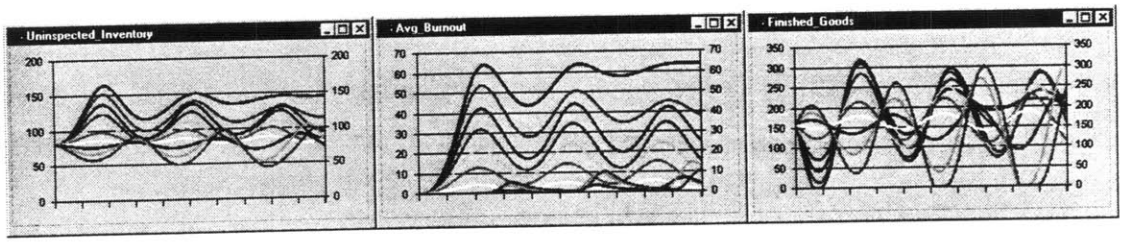

Figure 11: Program Behavior under a Sample Space Approximating Domain and a Sample-Aware Collecting Domain

Use of this new collected state domain reveals that higher inputs for *demand* appear to lead to not only higher amplitude cycles, but also to cycles of slightly longer periods. To formalize this observation, the user plugs in an additional collected semantic domain that accumulates each sample point and spectrally analyzes the time evolution associated with each sample space configuration. Rerunning the analysis under the new collecting domain, the user receives the output shown in Figure 12. The comparison between the results of two runs shows pronounced spectral differences between the operation of the program on one level of demand and on another.

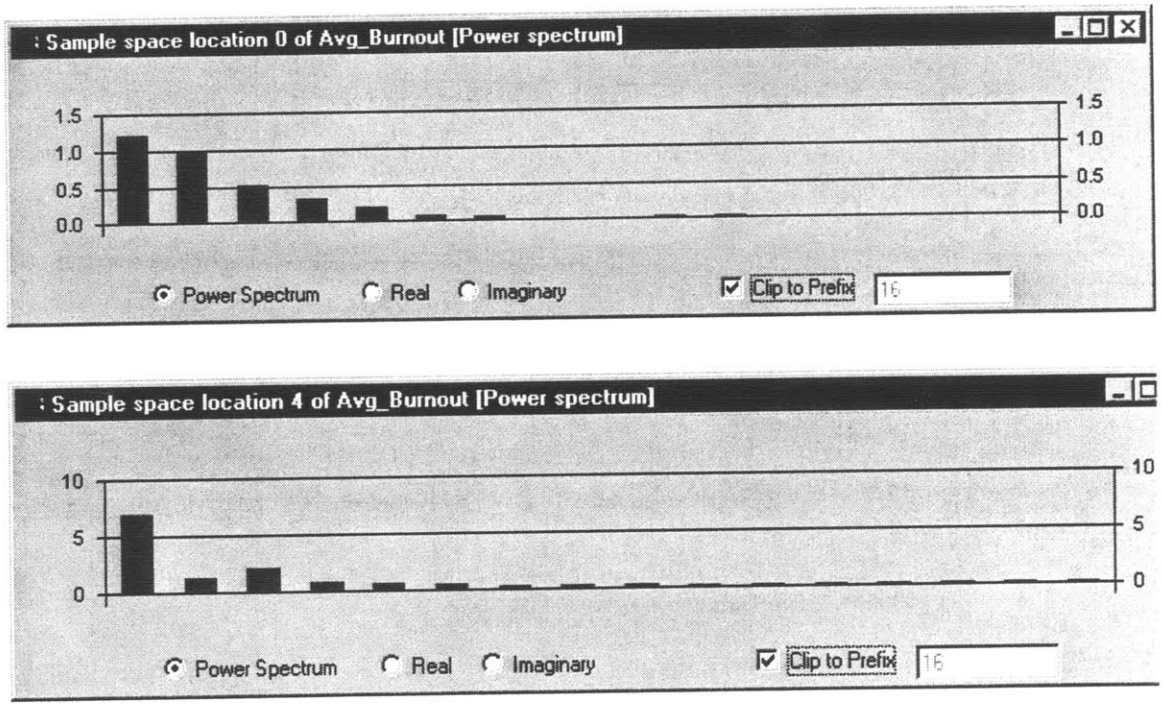

Figure 12: Comparison of output from the Spectral Analysis Collecting Domain for Two Different Demand Inputs.

Recall that the user was interested in understanding system behavior to a *distribution* of possible inputs. Plugging in another collecting domain allows the user to visualize a particular variable at a particularly point in time as bounded range of possible values or as a histogram (for those cases where sample space information is available). The user plugs in this collecting domain and reruns the analysis using the same (sample space) approximating domain. Figure 13 shows the distribution for the input variable *demand*, while Figure 14 illustrates the derived distributions induced on different variables in response to that input at a particular point in time. As suggested by the graph for the variable *Avg_Burnout*, the sample space domain can suffer from its use of a discrete approximation to a continuous distribution.

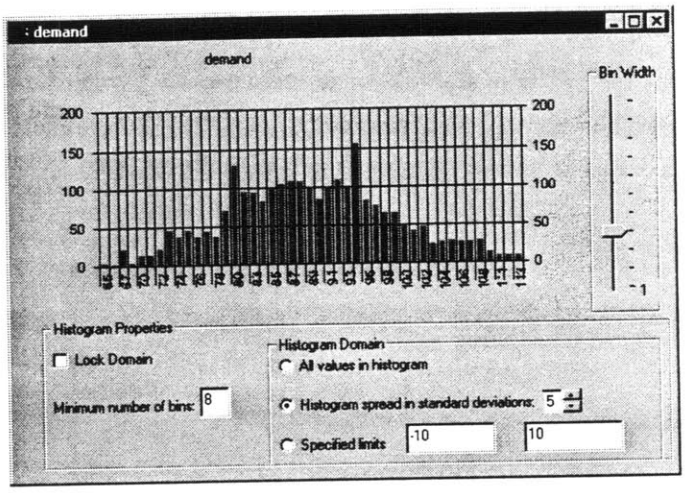

Figure 13: Input to the Program As a Distribution of Possible Values

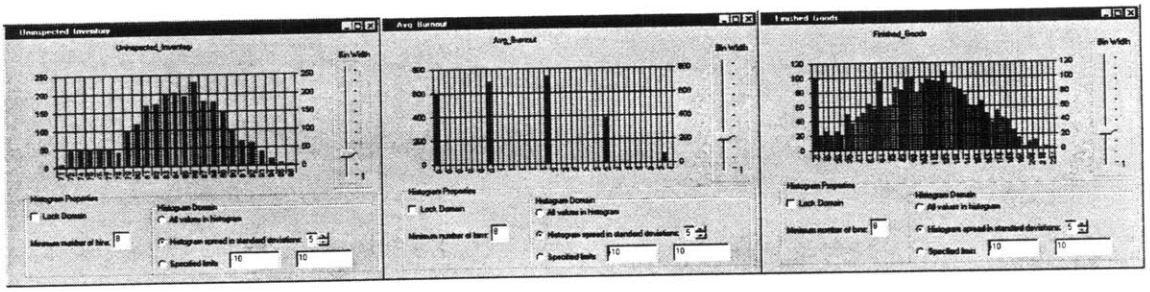

Figure 14: Response of Different System Variables at a particular time to the Distribution Given in Figure 13.

Finally, suppose that the user is interesting in trying to determine which conditions minimize the overall burnout within the company. Based on her observations of the program behavior depicted in Figure 11, she expects that the forced value for *demand* that minimizes this quantity would be that associated with the minimal change from the initial value for that variable (in effect, the external demand representing the least disturbance to the system). In order to verify this intuition, the user retains the same approximating domain, but plugs in a collecting domain that sums the values associated with the model variable *Avg_Burnout* over time. With this domain in place, the user reruns the analysis for two different lengths of time, and compares the results. To her surprise, the results differ: For shorter runs, a minimum change in *demand* from its original value indeed yields the least burnout in the workforce. But beyond a certain point, other factors dominate, and the configurations in sample space exhibiting the least burnout are those associated with the *lowest* overall demand (rather than those associated with the smallest departure from the original value for the variable).

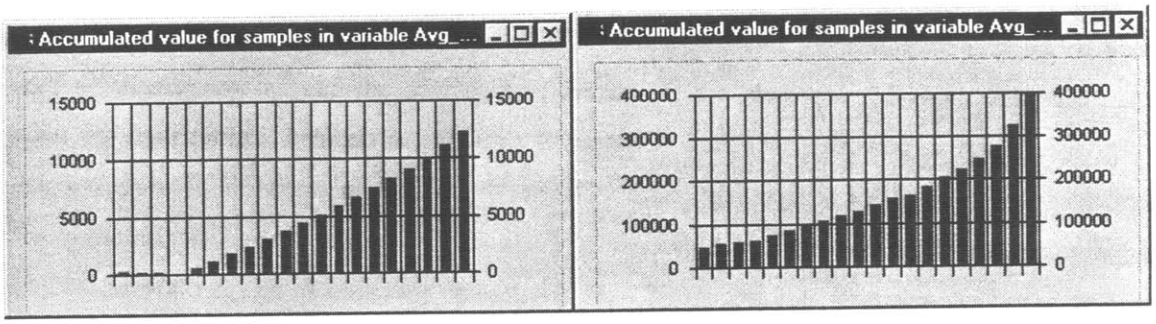

Figure 15: Comparison of Summing Collected Domain Results for Short (Left) and Long (Right) Runs.

1.4.5 Conclusion

This section has provided a brief, high-level example that illustrates the application of the "parameterizable generic analysis" approach to a specific example. While the particular example chosen is numeric, the same general issues and modes of interaction would be seen in the analysis of any type of program. Execution of the program under parameterizable abstract domains that approximate or collect information on program behavior provides a powerful but convenient means of gaining insight into program operation. As we shall see in Chapter 9, Chapter 10, and Chapter 11, this approach offers not only flexibility and choice to the user, but also greatly simplifies the construction of the analyses.

1.5 Roadmap of the Thesis

The previous section has provided a brief glimpse at the manner in which the pieces of TACHYON work together to allow a large variety of analyses to share the same analysis executable. The section included a look at the technique of compiling a fragment of source code to its "abstract semantics analogue", at the value and state interfaces used in the compiled code, at some sample value domains and the rough means by which they specify their semantic rules. Each of these pieces is discussed in detail within subsequent chapters.

The next chapter first introduces the reader to abstract execution, the formal characterization of program analysis that makes this thesis possible. Chapter 3 builds on the work of Chapter 2 in order to describe how we can create an extensible analysis system that allows for easy run-time customization of existing analyses by adding additional structure to the abstract domains. Up until this point, the focus has been on formal constructions rather than on engineering issues. Beginning with the architectural overview in Chapter 4, the thesis strikes an applied stance. Chapter 5 describes the software interfaces for abstract values through which extensible abstract execution takes place and which the user must define in order to create a new custom analysis. Chapter 6 describes analogous interfaces used for abstract states. The

engineering perspective continues in Chapter 7, where we discuss the motivations and background to the use of the "compiled analysis" approach, which garners performance advantages by specializing the analysis to operate on a particular program. Chapter 8 provides a detailed examination of two implementations of the compiled analysis approach. Chapter 9 provides an overview of some sample collecting domains that were created in the process of this work, with an eye towards providing an understanding as to the constraints and amount of work required to construct an abstract domain. This chapter includes a code generation domain that offers a simple means of creating the first run-time code generation strategy based on *online* specialization but one which appears too slow to be practical). Chapter 10 and Chapter 11 examine a set of sample approximating value domains. Chapter 10 examines approximating domains with fixed structure. Motivated by the desire to give users the option of comprehensively trading off the running time and precision of the analysis, Chapter 11 investigates two novel approximating value domains offering *scalable* precision. One of these domains is particularly noteworthy in that it achieves higher precision by representing values as disaggregated collections of values over a sample space of possible execution contexts. Finally, Chapter 13 summarizes the contributions of and issues raised by the thesis and lays out some avenues for future research.

SECTION I: FORMAL BACKGROUND

Chapter 2 Abstract Execution

Constructing a framework that offers the ability for the user to easily implement an arbitrary variety of custom program analyses requires a powerful and general-purpose approach to analysis. To this end, this chapter reviews the formalism of "abstract execution" that underlies many recent program analyses. An analysis based on abstract execution is characterized by the adherence to a formal approximation relationship to the familiar process we refer to as "running a program." As a result, the presentation in this section will of be a rather abstract and formal. The practically oriented reader is encouraged to wade through the symbolism, however, for it provides the basis for and insight into the concrete implementations discussed in later chapters.

2.1 Initial Intuitions

The common view of the source code of a program is as the specification of an algorithm that operates on particular, fully known (or "concrete") values. If we run this program in any specific execution context (i.e. for any fully specified set of external data), all values within the current state of the program will be completely known at any given point during the execution of the program. The text of each statement within the program describes how to map that (concrete) state to a new (concrete) state.[1] However, the same program text can be looked at from many different and less concrete perspectives.

Although we are not used to doing so, we can also think of the same source code as defining an algorithm on different categories of (concrete) values – for example, on values that are described only as "even"/"odd"/"unknown", or as "positive"/"negative"/"zero"/"unknown". The identical program text here can be thought of as a recipe that specifies how we calculate other values that are categorized in the same way, given certain external inputs that are broken up into these (disjoint) categories. Within this "abstract semantics", a given operator (say "+") will have different rules specifying its behavior (for example, if we are choosing to categorize values into "even" and "odd", "+" will be associated with the function specified in Figure 18. On the other hand, if we are dividing values up according to their sign, the "+" function is as shown in the Figure 17.) While a particular program operator will in general be associated with a somewhat different behavior within each abstract semantics, it is critical to understand that the assignment of meaning to the operator within a particular semantics is far from arbitrary.

[1] Note that we are speaking here of imperative languages, although many of the concepts that we discuss here can be applied directly to the realm of other languages. (Indeed, much of the early work in abstract execution took place in the context of functional and logic programming languages.)

Informally speaking, any operator must behave in its own domain in a manner that is "consistent" with – and an approximation to -- the behavior of the corresponding operator in the concrete domain.

This discussion has focused to this point on the description of values and the expressions that handle them. Program states can be described as structured collections of values and bookkeeping machinery, and can be characterized using similar principles.

In general, we can think of an *analysis* of program behavior as an "execution" of the program in which we associate some alternative information with program quantities, and during which information concerning the program execution is built up as the program "runs" in this approximate manner. Within an analysis, *some* abstract values and states simply serve as approximations to their run-time analogues; many, however, will typically serve as approximations to *sequences of* (or histories of) such quantities. Thus, analyses determining the set of "reaching definitions" for variable reads or the initialization status of a value would carry around different information approximating the sequences of operations with which a value has been computed – that is, different *abstractions* of this history information. A typical analysis would "save away" some of the information resulting from these computations. This collected analysis information might then be used in any number of different ways – in checking program assertions, for feedback to the user, for optimization, etc.

		Operand 2			
		0	1	2	...
Operand 1	0	0	1	2	...
	1	1	2	3	...
	2	2	3	4	...
	3	3	4	5	...
	⋮	⋮	⋮	⋮	⋮

Figure 16: Rules for Addition in the Standard Semantics

Operand 2 \ Operand 1		⊥	-	0	+	⊤
	⊥	⊥	⊥	⊥	⊥	⊥
	-	⊥	-	-	⊤	⊤
	0	⊥	-	0	+	⊤
	+	⊥	⊤	+	+	⊤
	⊤	⊥	⊤	⊤	⊤	⊤

Figure 17: Rules for Addition in the Abstract Sign Semantics

Operand 2 \ Operand 1		⊥	Even	Odd	⊤
	⊥	⊥	⊥	⊥	⊥
	Even	⊥	Even	Odd	⊤
	Odd	⊥	Odd	Even	⊤
	⊤	⊥	⊤	⊤	⊤

Figure 18: Rules for Addition in the Abstract Even/Odd Semantics

Having examined the connection between analysis and "abstract execution" in a non-standard semantics from an informal perspective, we turn to look at it in a more concrete and detailed manner.

2.2 Fundamentals

This section provides the conceptual underpinnings for the abstract execution process. Its primary function is pedagogy, as well as to introduce some of the notation used elsewhere in this chapter (and, to a lesser degree, in the thesis as a whole). The treatment is quite abstract at first (characterizing mathematical connections between two spaces), but becomes more applied thereafter.

2.2.1 Abstract Domains

The first step in formulating a program analysis as an abstract interpretation is the definition of the "categories" with by which we wish to classify program objects (for example, dividing values into "even/odd", "positive/zero/negative"), and the connections between those categories. These categories delineate the domain from which the analysis objects are drawn. Because they typically hide some details and expose only certain aspects of the actual concrete sets of values or states for which they stand, we will term the analysis values "abstract values" and "abstract states", and the domain from which they are drawn the "abstract domain". Following [Ayers 1993] we will denote objects drawn from the abstract domain with a superscripted "#" sign, as in $Value^{\#}$ and $State^{\#}$. Although the constructions apply equally well to both values and states, to keep the discussion generic we will term the abstract domain $S^{\#}$.

2.2.1.1 *The Structure of the Abstract and Concrete Domains*

For a particular abstract domain, we can define an "abstraction function" (or "upper adjoint") α that maps a particular *set* S of concrete objects (such as the integer values {1,2}, or a set of possible concrete states following a loop) into a *specific* corresponding abstract element that approximates the complete set of such values within the abstract domain (such as "positive", or an abstract state with any of a possible set of memory allocations). Informally, we term $\alpha(S)$ the "abstract representation" of S in the abstract domain. More informally yet, we can say that $\alpha(S)$ approximates concrete set S in the abstract domain.

It is important that $\alpha(S)$ is defined for all $\in \wp(S)$. Two particularly important cases to consider are $\alpha(\{\})$ and $\alpha(U)$ (where U is the universal set of *all* concrete objects). We denote $\alpha(\{\})=\bot$ and $\alpha(U) = \top$. At an informal level, we can think of $\bot$ as representing *no* object (e.g. state or value), and $\top$ as representing *any* possible object. Figure 19 shows a view of the abstraction function for the sign domain.

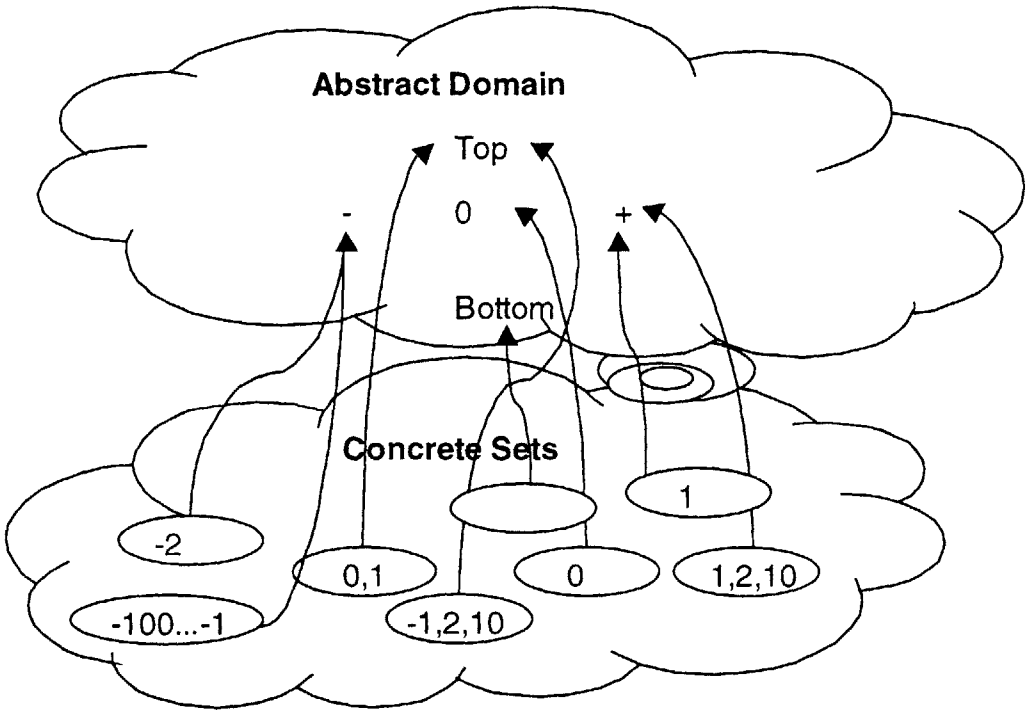

Figure 19: The Abstraction Function for the Sign Domain.

As suggested by Figure 19, the process of abstraction typically involves hiding (or ignoring) some details. In general we would expect the function $\alpha(S)$ to map *many* different sets S of concrete objects to the *same* particular element of the abstract domain (for example, we expect that for an analysis that classifies

sets of values into "negative", "zero", "positive", or "unknown", there would be many different sets of values mapped onto "positive"). Because it is typically not an injection, $\alpha(S)$ will in general lack a true inverse: For example, we cannot map back from "positive" to a unique set of concrete objects (values) that are indicated by it. Nonetheless, given $\alpha(S)$: $2^S \rightarrow S^\#$, we can define a near inverse of $\alpha(S)$ that is historically called the "concretization function" (or "lower adjoint"). We denote this function as $\gamma(a^\#)$; it is associated with functionality $S^\# \rightarrow \wp(S)$. This near inverse gives the maximal subset of S that maps to the abstract element $a^\#$; for example, in the sign domain, $\gamma(positive)=\{1,2,3,\ldots, MaxInt\}$.

The abstract domain $S^\#$ is not merely an unordered collection of elements; its relationship with the domain S through the abstraction function α induces a very important structure upon it. In particular, we can impose a partial ordering on the abstract value domain that corresponds very intuitively to the level of approximation of different elements. Given two elements a and b $\in S^\#$, we write a $\sqsubseteq$ b (informally, "b approximates a") if $\gamma(a) \subseteq \gamma(b)$ (informally, b abstractly represents at least all of the concrete sets abstractly represented by a). In other words, if one element b is located higher in the abstract domain than another element a, we can think of b as representing a more general class of values than are associated with a. Because $\gamma(\top) = \mathbf{U}$ and $\gamma(\bot) = \{\}$, $\forall a \in S^\#$, $\bot \sqsubseteq a$ and $a \sqsubseteq \top$. Similarly, we can define a "least upper bound" operator $\sqcup$: $S^\# \times S^\# \rightarrow S^\#$ that identifies the least element of the abstract domain that safely approximates both of the two arguments (also themselves elements of the abstract domain). (More formally, given a,b $\in S^\#$, $a \sqcup b = c \in S^\#$ | $a \sqsubseteq c$ & $b \sqsubseteq c$ & ($\forall$d such that $a \sqsubseteq d$ & $b \sqsubseteq d$, $c \sqsubseteq b$)).

As might be expected from the structure of the domains, the two adjoints α and γ satisfy a number of mathematical properties, as listed below. The first two properties apply individually to the adjoints, while the second two are joint relationships that guarantee a certain sense of "compatibility" between the adjoints:

Monotonicity of α. Given $S_1, S_2 \in \wp(S)$ and $S_1 \subseteq S_2 \Rightarrow \alpha(S_1) \sqsubseteq \alpha(S_2)$

Monotonicity of γ. Given $s_1, s_2 \in S^\#$ and $s1 \sqsubseteq s_2 \Rightarrow \gamma(S_1) \subseteq \gamma(S_2)$

Extension: $\forall S \in \wp(S)$, $S \subseteq \gamma(\alpha(S))$. Intuitively speaking, we can think of this as indicating that while we may lose information concerning the original set S by mapping it to the abstract domain, we will

always do so in a way that is conservative: Namely, by using an abstract representation $\alpha(S)$ that actually represents a larger set of concrete objects than we need to. (For example, if S = {1,2}, for the sign domain $\alpha(S) = positive$, and $\gamma(\alpha(S)) = \{1,2,\ldots, MaxInt\}$.)

Reduction: $\forall a^{\#} \in S^{\#}$, $\alpha(\gamma(a^{\#})) \sqsubseteq a^{\#}$. Less formally, this means that if we start with an abstract value $a^{\#}$ (e.g. "*Even*" in the Even/Odd domain), it is guaranteed to be a valid approximation to $\gamma(a^{\#})$, the set of concrete objects that it represents (for the Even/Odd domain, $\alpha(\gamma(\text{Even})) = \text{Even}$.).

From the properties above, it can be shown that given one adjoint the above conditions uniquely specify the other adjoint, and that the adjoints map lower bounds in one domain to lower bounds in the other.

2.2.1.2 Characteristics of Functions in the Two Domains

We have spoken above of how the individual elements of the abstract domain $S^{\#}$ can be thought of as approximations to sets of objects (e.g. values and states) in the concrete domain S. The connections between the concrete and abstract domains established by α and γ allow us to straightforwardly extend this relationship to *functions* operating in each domain. For instance, in Figure 17 and Figure 18, we illustrated the result of "adding together" two abstract elements of the Sign domain and the Even/Odd domain. We can now begin to formalize the relationship between functions and their abstract analogues.

In particular, given a function f: $S \rightarrow S$ that operates on values, we seek to define a corresponding function $f^{\#}$: $S^{\#} \rightarrow S^{\#}$ which approximates f in the abstract domain. In order to serve as abstract analogue to f, $f^{\#}$ here must satisfy a very particular set of properties. In particular, when we apply $f^{\#}$ to combine two objects a and b in the abstract domain, the result we obtain from the operation must be a valid approximation to (consistent with) the result of performing the operation f in the concrete domain. A bit more formally, given f: $S \rightarrow S$, $f^{\#}$: $S^{\#} \rightarrow S^{\#}$ $f^{\#}$ forms a legal approximation to f if $\forall a^{\#} \in S^{\#}$, $f_S(\gamma(a^{\#})) \subseteq \gamma(f^{\#}(a^{\#}))$. (Note here that f_S is f extended to handle *sets* of concrete objects rather than concrete objects. That is, f_S: $\wp(S) \rightarrow \wp(S)$, where $f_S(S) = \{ f(v) \mid v_1 \in S \}$).

The above definition has sharpened our definition of what it means for an abstract function $f^{\#}$ to approximate a concrete function f, but leaves great latitude in the choice of a $f^{\#}$ -- *there* are many possible $f^{\#}$ that satisfy the constraints above. For example, $f^{\#} = \lambda a. \top$ would represent a valid (but rather uninteresting) approximation to *any* function f. Depending on the abstract domain and function at hand, there are likely to be many other abstract functions that would serve as allowable abstractions of f as well. The properties of the adjoints above, however, can be used to demonstrate that for any concrete function

f, there exists a maximally precise[2] abstract function $f^{\#}$ that emulates f in the abstract domain. This function $f^{\#}$ has a particularly simple definition: $f^{\#}(a^{\#}) = \alpha(f_S(\gamma(a^{\#}))$.

2.2.1.3 Abstract Values and Abstract States

The section above has characterized Galois Connections, and described some of the properties associated with such connections. As suggested by the occasional example, the approximation of run-time values and states can be described as Galois connections. In particular, for a particular approximation of program behavior, program values and program states will each be associated with pairs of adjoints. These adjoints describe the manner in which run-time values are represented by the approximation. As suggested by the mathematics of Galois connections, the approximations employed typically suffer from information loss. This reflects the fact the fact they "clump together" many distinct run-time states or values into the same approximate abstract analogue (e.g. by characterizing any subset of positive integers as "positive", and any subset of abstract states with certain numbers of allocations from a particular allocation site by a single abstract state with multiple pointers to a single abstract location). The discussion above recognized the formal basis for the approximation of values with abstract values and of states with abstract states, but also discussed the approximation of functions on values by functions on abstract values and functions on states by functions on abstract states. The section below builds on this understanding by discussing members of a very particular class of function on states (*interpretive* functions), and their abstract analogues (abstract interpretive functions, or abstract interpreters). Like abstract values and abstract states, such functions play central roles in program analysis.

2.3 Execution in the Concrete and Abstract Domains

2.3.1 Introduction

The discussion to this point has focused on simple formalisms: The character of program values and states and their analogues in an abstract domain. We have also discussed how functions within the concrete domain can be approximated in an abstract domain. This section makes use of these earlier discussions to look at how *execution* in an abstract domain relates to execution in the concrete domain. Because we are particularly interested in abstract executions that carry out *analysis* of program behavior, our comments will focus primarily on executions in which we "accumulate information" on the course of interpreting the program in the abstract domain. While this discussion may seem unnecessarily abstract

[2] Note that the definition of the intuitive notion of "maximally precise" here is simply based on the extension of the ordering relation $\sqsubseteq$ of the abstract domain pointwise to functions. That is, given two functions $f_1^{\#}, f_2^{\#} : Value^{\#} \rightarrow Value^{\#}$, $f_1^{\#} \sqsubseteq f_2^{\#}$ if $\forall\, a \in Value^{\#}$, $f_1^{\#}(a) \sqsubseteq f_2^{\#}(a)$. Given two such functions $f_1^{\#}, f_2^{\#}$, where both $f_1^{\#}$ and $f_2^{\#}$ approximate concrete function f, we can refer to $f_1^{\#}$ as a more precise approximation to f if $f_1^{\#} \sqsubseteq f_2^{\#}$.

or mathematical, it formulates the framework we will use for the very practical analyses presented in later chapters within this thesis.

There are many ways to describe the semantics of a program text. For the purposes of this chapter, we will focus on one specific approach: Through the operation of an "interpreter" *I*. In this section, we will survey several interpreters, each associated with different – but closely related – semantics and each offering a different view of program operation.

We began our discussion of the semantics of values and states with an understanding of how such objects are manipulated in "normal program operation" (that is, operation under the standard semantics). We will follow the same tack here, and proceed on to a discussion of program operation under the types of abstract semantics that are of interest during program analysis.

2.3.2 The Standard Semantics Interpreter

Within an imperative language, program operation proceeds as a series of steps from state to state. For each step, a program construct (a "statement" or "command") changes some aspect of the state (even if just the instruction pointer). We can describe this behavior with a "interpretation function" I: *State* $\rightarrow$ *State*. It is worth noting in passing that the function I is specific to the particular program being interpreted. That is, I has no capacity for parameterization by the program of interest, and must have this those "hardcoded" within its definition.[3]

The initial state for I is termed s_0. A "halt" state $s_{i,\infty}$ is a state in which that $I(s_{i,\infty}) = s_{i,\infty}$. (Note that we subscript the state $s_{i,\infty}$ in order to communicate the fact that there may exist more than one such state for a given interpreter I.) The execution sequence of a program can be characterized as the (possibly infinite) set of states $\cup\{ I^i(s_0) \}\ \forall i \geq 0$.

2.3.3 The Abstract Interpreter

In Section 2.2.1.3, we noted that the concrete domain *State* could be approximated by an abstract domain *State*$^\#$. The two partially ordered sets are linked by a Galois connection with abstraction function α_S and concretization function γ_S. In accordance with the earlier discussions, a function f : *State*$\rightarrow$*State* operating over the concrete domain can be approximated by a set of different functions $f^\#$: *State*$^\#\rightarrow$*State*$^\#$. So we know that the interpretation function I has one or more correlates $I^\#$ in the abstract domain, where each $I^\#$ approximates the behavior of I within the abstract domain to different levels of approximation.

---

[3] In this sense, the function definition formally anticipates the "compiled analysis" strategy described in Chapter 7 and Chapter 8.

As discussed in Section 2.1, executions with abstract domains can be informally viewed as performing *analyses* of the program, and we are evidently quite close to our goal of sketching a general conceptual framework for describing analyses of program behavior.

However, we must still make one significant change in our understanding of the way in which analysis relates to program execution. In particular, Section 2.1 noted the importance to analysis of execution with domains which accumulate some sort of "history information" to be accumulated and maintained over the course of the analysis. For example, an analysis conducted for the sake of optimization might accumulate information concerning variable use over time in order to deduce if a particular variable is always associated with a particular concrete value; an analysis performing reference counting on objects manipulated by the code might maintain conservative reference count estimates on those objects while scanning the code; a profiling-oriented analysis would accumulate information on the number of times a given loop is executed or the number of entry points to a particular function, and a code coverage analysis would record the basic blocks through which execution has traveled.

None of this "accumulated" sort of information has a place within the abstract domains defined above: The value abstraction function $\alpha_v : \wp(Value) \rightarrow Value^{\#}$ is defined so as to approximate *values* (rather than value histories), and the state abstraction function $\alpha_S: \wp(State) \rightarrow State^{\#}$ provides similar functionality in the state domain. While interesting from a conceptual standpoint, such an abstract execution would not allow for the collection of information on program behavior. Instead of the analysis operating on abstractions of values and abstractions of states, we seek an interpreter operating on abstractions of value *sequences* and state *sequences*. The next section describes the formalizations that accomplish this.

2.3.4 The Abstract Collecting Interpreter

To accord with the fact that the information collected during analysis can depend on the entire span of the (abstract) execution of a program, we define a "collecting interpreter" I_C. I_C operates in a manner very similar to I, but accumulates a complete record of each state it passes through. Thus, $I_C : \wp(State) \rightarrow \wp(State)$ has the definition $I_C(S) = S \cup \{ I(s) \mid s \in S \}$. It is straightforward to show that $I_C^n(\{s_0\}) = \cup\{I^i(s_0) \mid 0 \leq i \leq n \}$. (Put more informally, the result of applying I_C n times is simply the accumulation of the states produced at each step of applying I n times.)

Given the collecting interpreter I_C, we can formulate a corresponding abstract interpreter that maps abstractions of state sequences to abstractions of state sequences. In essence, each such state sequence represents a trace of program behavior, and the abstraction function embodies the rules used to distill the

desired information concerning program behavior from the program traces. At an intuitive level, a given abstract state sequence approximates many possible program traces, and an abstract value many possible value computations.

In order to characterize the abstract interpreter operating on accumulated information, we define our abstract state domain *StateSequence*$^{\#}$ and abstract value domain *ValueSequence*$^{\#}$, and define Galois mappings with $\alpha_{C,S}$: $\wp(State) \rightarrow$ *StateSequence*$^{\#}$, $\gamma_{C,S}$: *StateSequence*$^{\#} \rightarrow \wp(State)$, as well as $\alpha_{C,V}$: $\wp(Value) \rightarrow$ *ValueSequence*$^{\#}$, $\gamma_{C,V}$: *ValueSequence*$^{\#} \rightarrow \wp(Value)$. Based on this Galois mapping and the interpreter I_C, we can build on the discussion of function approximation in Section 2.2.1.2 and define an abstract collecting interpreter $I_C^{\#}$: *StateSequence*$^{\#} \rightarrow$ *StateSequence*$^{\#}$. As discussed within the last section, one particularly precise (but very expensive) choice for $I_C^{\#}$ would be to set $I_C^{\#} = \alpha \bullet I_C \bullet \gamma$.

Note here that while the upper and lower adjoints at hand are associated with the same functionalities (modulo labeling) as when we created the interpreter $I^{\#}$, the semantics involved is rather different. This reflects the fact that while the *State*$^{\#}$ domain associated with $I^{\#}$ abstracted specific states or sets of states, the *StateSequence*$^{\#}$ domain here abstracts the properties of *sequences* of states. Similarly, while *Value*$^{\#}$ approximated possible run-time values, the value domain *ValueSequence*$^{\#}$ approximates possible *sequences* of run-time values (in particular, the approximations to values used to compute the current element of *ValueSequence*$^{\#}$). While the sets of states in the domain of α_S exhibited no necessary ordering, the sets of states in the domain of α_C are expected to be composed of a set of legal *execution sequences* of states. That is, for any $\alpha_C(S)$, given a set of possible start states S_0, we would expect $S = \cup I_C^n(\{s_0\}) \forall s_0 \in S_0 = \cup(\cup\{I^i(s_0) \mid 0 \leq i \leq n\}) \forall s_0 \in S_0$.

In short, an abstract interpreter $I_C^{\#}$ "single steps" a program by "executing" one statement at a time. But instead of mapping a state to another state, each such statement execution maps an approximation to the possible sequence of states up until the *current* statement into an approximation of the possible sequence of states up until the *next* statement. When one obtain information from an analysis of program behavior, the data represents an abstraction of the sequence of states through which the program travels. $I_C^{\#}$ thus represents a general-purpose model of a program analyzer, where the character of the data that is collected and the rules for collecting it are left unspecified. In order to implement a particular program analysis, the abstract domains *ValueSequence*$^{\#}$ and *StateSequence*$^{\#}$ would have to be defined, and the upper adjoint α_C and an interpretation function $I_C^{\#}$ would have to be specified. This function $I_C^{\#}$ would encode the rules for collecting the information accumulated within the abstract *StateSequence*$^{\#}$.

2.3.5 *The Analysis Process*

2.3.5.1 *Introduction*

We have noted that an abstract "collecting" interpreter $I_C^{\#}$ appears to serve our general needs for modeling the collection of analysis information on program behavior, but have not yet discussed the manner in which the analysis actually takes place. One item of particular concern is related to an important difference between the standard and collecting semantics on the one hand and the needs of analysis on the other. In particular, while the sequence of states $I_S^n(s_0)$ generated by a standard-semantics program may be infinite in length, we seek to be able to ensure that an analysis of that program's behavior will terminate within a finite (and hopefully more tightly bounded) amount of time. This section examines the analysis process.

2.3.5.2 *Existence of Fixed-Points*

In presenting the interpreter I_S, we noted the existence of "halt" states $s_{i,\infty}$ distinguished by the fact that $I_S(s_{i,\infty}) = s_{i,\infty}$. For a given halting computation beginning with start state s_0, we can think of the "result" of the computation as being the state s_∞ such that $I_S^n(s_\infty) = s_\infty$. Informally, we can write that $s_{i,\infty} \in \mathbf{Fix}(I_S)$, where "**Fix**(*f*)" is the set of fixed points of function *f*. To abuse the notation somewhat, we can write more specifically that $s_{i,\infty} \in \mathbf{Fix}(I_S, s_0)$. This expresses the fact that $s_{i,\infty}$ is a very particular fixed point of I_S, in particular one that is dictated by start state s_0. Note again that the existence of $s_{i,\infty}$ is by no means guaranteed (for instance, if the program encapsulated by the interpreter I_S always diverges and never halts, no such $s_{i,\infty}$ exists), and that there can in general be many fixed points of *f*.

We will take a similar (but more formal) least-fixed-point approach to defining the "result" of the analysis function, as applied to a particular initial state. Our first step in this direction will be to examine the result of executing the collecting semantics interpreter. Recall that I_C was defined as $I_C : \wp(State) \rightarrow \wp(State) = (\lambda S.\{I_S(s) \mid s \in S\})$. We obtain the result of executing I_C on a start state $\{s_0\}$ by using the Kleene fixed point theorem. In particular, given that $I_C^n(\{s_0\}) = \cup\{I^i(s_0) \mid 0 \le i \le n\}$ we can show that

$$\mathbf{Fix}(I_C, S_0) = \cup(I_C^n(\{s_0\})\ \forall\ s_0 \in S_0) = \cup\{I_S^n(s_0)\ \forall\ s_0 \in S_0\}$$

$$\mathbf{Fix}(I_C, S_0) = \{\text{ States in the execution sequences } s_0 \ldots \mathrm{Fix}(I_S, s_0)\ \forall s_0 \in S_0\ \}$$

In other words, the result of a executing collecting semantics interpretation of a program on a particular input – if it exists – consists of the "trace" of states $I_S^n(\{s_0\})$ up until the first point at which a state $s_{i,\infty}$ is reached. The result of executing that interpretation on a set of particular possible inputs is just the union of the traces of states resulting from executing the program on each program input. Note that because

there is no guaranteeing that a program will terminate in a finite time, the **Fix**(I_C,S_0) may well be of infinite cardinality.

The last section presented an abstract interpreter $I_C^{\#}$: *StateSequence*$^{\#}$→ *StateSequence*$^{\#}$ whose behavior approximated that of I_C, but which operated on finitely sized *abstractions* of state sequences and value sequences rather than on actual collections of states and values. The domain $\wp$(*State*) and *StateSequence*$^{\#}$ are connected by a Galois connection with a lower adjoint α_C: $\wp$(*State*)→ *StateSequence*$^{\#}$ which mapped a given state sequence S into an approximation α_C(S). Section 2.2.1.1 noted that lower adjoints map lower bounds in the concrete set domain to lower bounds in the abstract domain. Recalling this, we are in a position to describe an abstract approximation to the result of executing the collecting semantics interpreter I_C on a particular set of input states S_0. In particular, the most accurate analogue to **Fix**(I_C, S_0) in the *StateSequence*$^{\#}$ domain can be written as α_C(**Fix**(I_C, S_0)) = α_C($\cup I_S^n$({s_0} $\forall$ $s_0 \in S_0$)).

It is straightforward to show that because the abstract collecting interpreter $I_C^{\#}$ represents an approximation to the (concrete) collecting interpreter I_C, this abstract corollary of the fixed point of I_C is itself guaranteed to represent a fixed point of $I_C^{\#}$. In other words, $I_C^{\#}$(α_C(**Fix**(I_C, S_0)) = α_C(α_C(**Fix**(I_C, S_0)).

We have thus shown that given a fixed point of the collecting semantics interpreter, we can demonstrate the existence of – and constructively define – a fixed point for any sound *analysis* based on the corresponding abstract interpreter. While this may be encouraging at the conceptual level, at another level, it is trivial indeed, and quite useless – in essence, the equivalence tells us little more than the fact that a maximally precise analysis result can be deduced from a complete trace of the program in the standard semantics. It would be disheartening indeed if in order to perform an analysis based on abstract interpretation we were required to compute and accumulate the entire sequence of concrete states in the execution of a program – which is what is required to compute α_C(**Fix**(I_C, S_0)). For any non-trivial program this would require at the least a massive amount of storage, and would run the obvious risk of never converging (and thus lacking a well-defined analysis result). Fortunately, another approach that can provide us with a solution in a much more efficient and cost-effective manner.

2.3.5.3 Performing Analysis with the Abstract Interpreter

In order to understand the basis for an analysis exhibiting lower cost, we begin with the observation of Section 2.2.1.1 that a lower adjoint α: $A \rightarrow B$ of a Galois connection maps least upper bounds in A to least upper bounds in B. That is, given $a_i \in A$, $\alpha(\sqcup_A a_i) = \sqcup_B \alpha(a_i)$. (Note that here $\sqcup_D$ symbolizes the least

upper bound operator in domain D.) For the particular case of $\alpha_C : \wp(State) \rightarrow StateSequence^{\#}$, the $\sqcup_A$ operator corresponds to set union in the $\wp(State)$ domain, and we can write

$$\alpha_C(\mathbf{Fix}(I_C, S_0)) = \alpha_C(\cup I_S^n(\{s_0\} \ \forall \ s_0 \in S_0)) = \sqcup \alpha_C(I_S^n(\{s_0\} \ \ \forall \ s_0 \in S_0)).$$

It is worth pausing to consider this result. We had earlier shown that the maximally precise analysis of program operation for an abstract collecting domain was given by the expression $\alpha_C(\mathbf{Fix}(I_C,S_0))$ – a result obtained by first accumulating the complete set of possible traces of program execution, and distilling the entire set down to the desired analysis results through the application of α_C. The equality above suggests an incomparably cheaper manner of computing the same information. Rather than deriving all possible paths of execution in the standard semantics and then applying the abstraction function, the result of abstracting the fixed point of the collecting interpreter I_C can be built up *incrementally* by successively taking the least upper bound of the abstraction of each successive state produced by running the standard semantics interpreter I_S on the different inputs s_0. This is an encouraging discovery indeed, for it suggests that performing an abstract interpretation does not require the encyclopedic accumulation of the set of all states produced by the standard semantics interpreter. Instead, at any point during the analysis we can get away with only maintaining a single concrete state ($I_S^i(\{s_0\})$), and a single abstract state (representing the least upper bound of the abstractions of all previous states, or $\sqcup_{0 \le j \le i} I_S^j(\{s_0\})$) for each $s_0 \in S_0$, and yet obtain the same result. Each step of the analysis simply requires the application of α_C to each of the current state approximations and taking the least upper bound of the result of the abstraction with the accumulated abstract states. The concrete states can then be advanced to the next states. In essence, we simply step the program along in the *concrete* semantics, abstracting each concrete state in turn and accumulating the result of this abstraction. For the case where S_0 is small (or, better yet, a singleton set), this represents a dramatic cost savings over the naïve approach. What is particularly remarkable is the fact that there is no loss of precision involved. An additional advantage arises from the fact that the process uses only the standard semantics interpreter I_S and the abstraction function α_C – the implementation of an additional explicit interpreter (such as $I_C^{\#}$) is not required, as it is in many other approaches. Many existing run-time analysis systems make use of this powerful approach to perform analysis on a program on a single input at a time without requiring the use of any additional interpretive machinery. Such systems instrument the program of interest to collect information concerning its operation without changing its observable behavior in any salient way[4].

[4] Note that this could change the timing associated with the program, but not the behavior typically characterized as program extension.

While this analysis technique permits collection of precise information on program behavior within a general analysis framework, it does have several significant downsides. Two of the most important are considered here.

- **Potential for analysis non-termination.** If the standard semantics interpreter I_S never terminates on input $\{s_0\}$, the computation of the least upper bound in the abstract domain will either never terminate or will fail to collect information on the entire span of program execution.[5]

- **Inability to Analyze Program Behavior on many possible Inputs.** The strategy discussed above requires maintaining an abstract collected state and a concrete state for *each possible starting state* s_0. Because starting states are typically associated with different patterns of program input, this requires the analysis to separately simulate the program on each possible input. For cases where only a single input is to be considered, this approach is very cheap. Unfortunately, this strategy rapidly becomes infeasible as size of the possible input sets increases. This constraint is reflected in the fact that contemporary run-time analysis systems run only on a single input at a time.

If we are willing to live with the constraints that the termination of an analysis depends upon that of the program being analyzed and that analyses may be run only on a particular input pattern, the above strategy seems attractive. However, there are many software systems whose termination is *not* guaranteed but which form attractive targets for analysis. Operating systems and calculations of quantities to arbitrary precision and a variety of application software are obvious examples of this class of software. In certain cases, the potential for continuous execution makes analysis yet more desirable: For instance, it may be particularly important to apply analysis aimed at eliminating potential run-time error conditions, or to optimize program performance. While the desire to provide the user the option of guaranteed analysis termination is a general one, in the context of such programs it is particularly acute. We now turn to discuss a method by which analysis termination can be ensured, which can drastically reduce the number of interpreter steps required to yield the complete analysis results, and which makes feasible the simultaneous analysis of a program on many possible inputs. Such benefits come with a heavy cost of their own, however: Significantly reduced analysis precision.

---

[5] It is critical to realize here that because the computation $I_S^n(s_0)$ is taking place using the *standard* semantics, even if a point is reached where $\alpha_C(I_S^n(\{s_0\})) = \alpha_C(I_S^{n-1}(\{s_0\}))$, it is in general not safe to terminate the analysis: It may well be the case that $\alpha_C(I_S^{n+1}(\{s_0\})) \neq \alpha_C(I_S^n(\{s_0\}))$. As a result, unless $\alpha_C(I_S^n(\{s_0\})) = \top$ or $\bot$ or $I_S^n(\{s_0\}) = I_S^{n-1}(\{s_0\})$ for some n, it is not possible to terminate the analysis. Note importantly that the potential for analysis non-termination remains even if the *StateSequence*$^\#$ domain is of finite height: Because I_S^n is driving the analysis, the least upper bound operator could be applied arbitrarily many times before the height of the result is increased in the *StateSequence*$^\#$ domain.

2.3.5.4 *Guaranteeing Analysis Termination*

2.3.5.4.1 *Introduction*

As was noted earlier, the strategy employed above for performing analysis though calculation of $\sqcup\alpha_C(I_S^n(\{s_0\}\ \forall\ s_0 \in S_0))$ makes no use of an explicit interpreter $I_C^\#$: The analysis is instead carried out using only the standard semantics interpreter I_S. While this allows realization of maximal analysis precision, it forces the analysis to take place at a pace dictated by the standard (concrete) semantics. A faster strategy would take advantage of the capacity to conduct *abstract* rather than concrete interpretation, in essence benefiting from the mapping of many concrete steps into a single "abstract" step. We explore that approach below.

2.3.5.4.2 *Defining the Abstract Interpreter*

We noted above that given a standard semantics interpreter I_S, we could define a "maximally precise" abstract interpreter $I_C^\#(s^\#)=\lambda S.\alpha(I_S(\gamma(s^\#)))$. Following [Ayers 1993] we express this in functional composition notation as

$$I_C^\#=\alpha_C \bullet I_S \bullet \gamma_C.$$

Composing the above with α_C we arrive at $I_C^\# \bullet \alpha_C=\alpha_C \bullet I_S \bullet \gamma_C \bullet \alpha_C$. According to the properties of a Galois Connection between $\wp(A)$ and B, $S \subseteq \gamma(\alpha(S))$. This implies that $\alpha_C \bullet I_S(S) \sqsubseteq \alpha_C \bullet I_S \bullet \gamma_C \bullet \alpha_C(S)$. But we can use the definition of $I_C^\#$ above to rewrite this as

$$\alpha_C \bullet I_S \bullet \gamma_C \bullet \alpha_C(S)=I_C^\# \bullet \alpha_C$$

and thus

$$\alpha_C \bullet I_S \sqsubseteq I_C^\# \bullet \alpha_C.$$

In other words, the result of applying the standard semantics interpreter I_S to a state and abstracting is guaranteed to be safely approximated by the result of first abstracting the state and then applying the abstract interpreter $I_C^\#$ to the resulting *StateSequence*$^\#$.

While equation says nothing about how much information will be lost in the approximation, it does give some latitude in interchanging an occurrence of I_S with an occurrence of $I_C^\#$. Phrased in another way, this hints that while the *most accurate* analysis may be obtained by interpreting using I_S and finally abstracting, a less accurate – but potentially faster – solution can be arrived at by first abstracting the state and then interpreting using an (explicitly constructed) abstract interpreter $I_C^\#$. For a single step, this

would probably save no time at all, but if this is done for *many* steps, very significant cost savings can result.

Note that the equation $\alpha_C \bullet I_S \sqsubseteq I_C^{\#} \bullet \alpha_C$ implies $\alpha_C \bullet I_S \bullet \gamma_C \sqsubseteq I_C^{\#}$ (by the Reduction property of Section 2.2.1.1). This provides an important and intuitive constraint on the operation of $I_C^{\#}$: *In order to constitute a correct abstract interpretation, the result of abstracting a state and stepping forward in the abstract interpreter must approximate the result of just stepping that state forward in the standard semantics interpreter and then abstracting.* It is this "approximate commutativity" of abstracting and interpreting that serves as the fundamental soundness constraint for this thesis.

Given that $\alpha_C \bullet I_S \subseteq I_C^{\#} \bullet \alpha_C$, we can take advantage of the fact that $S \subseteq \gamma(\alpha(S))$ and perform one more composition on the left with γ_C. This yields an approximative definition for I_S in terms of $I_C^{\#}$: $I_S \subseteq \gamma_C \bullet I_C^{\#} \bullet \alpha_C$.

2.3.5.4.3 Reasoning about Abstract State Sequences

We now recall the formula $\alpha_C(\mathbf{Fix}(I_C, S_0)) = \sqcup \alpha_C(I_S^n(\{s_0\} \ \forall \ s_0 \in S_0))$. Substituting in the formula above for I_S and taking advantage of the monotonicity of α, we have

$$\alpha_C(\mathbf{Fix}(I_C, S_0)) \sqsubseteq \sqcup \alpha_C((\gamma_C \bullet I_C^{\#} \bullet \alpha_C)^n(S_0)).$$

At an intuitive level, this accords with the commonsense fact that the result of executing an abstract interpreter on each successive concrete state through which a program may travel must safely approximate the result of directly abstracting the full digest of an execution sequence.

Now because $S \subseteq \gamma(\alpha(S))$

$$(\gamma_C \bullet I_C^{\#} \bullet \alpha_C)^n \subseteq \gamma_C \bullet (I_C^{\#})^n \bullet \alpha_C$$

Again using the monotonicity of α_C, we know that

$$\alpha_C((\gamma_C \bullet I_C^{\#} \bullet \alpha_C)^n) \sqsubseteq \alpha_C(\gamma_C \bullet (I_C^{\#})^n \bullet \alpha_C).$$

This approximate equality demonstrates that a system in which each successive set of possible states are maintained as sets of standard semantics states (and are only mapped to abstract states for stepping forward at each point) is safely approximated by a much cheaper system in which the *initial* state is

mapped to the abstract domain, and all subsequent operation takes place in the abstract domain. Note again, however, that some information may be lost in so doing.

2.3.5.4.4 Deriving the Result of Executing a Program in the Abstract Semantics

Recall that the goal of performing abstract execution here is the derivation of the *result* of a program in the abstract semantics (presumably some collected information on program behavior).

Noting that given $a_i \sqsubseteq b_i \mid 0 \leq i \leq n$, $\sqcup a_i \sqsubseteq \sqcup b_i$.

we can use the two approximate equalities above to demonstrate that

$$\alpha_C(\mathbf{Fix}(I_C, S_0)) \sqsubseteq \sqcup \alpha_C((\gamma_C \bullet (I_C^{\#})^n \bullet \alpha_C)\ (S_0)).$$

Since $\alpha_C(\gamma_C(s^{\#})) \sqsubseteq s^{\#}$, we can further write that

$$\alpha_C(\mathbf{Fix}(I_C, S_0)) \sqsubseteq \sqcup (I_C^{\#})^n(\alpha_C(S_0)).$$

In other words, we are guaranteed to obtain a valid approximation to $\alpha_C(\mathbf{Fix}(I_C, S_0))$ – the result of abstracting the full digest of states through which the program can pass – by first abstracting the set of possible initial states to a *StateSequence*$^{\#}$ and then successively applying $I_C^{\#}$ and taking the least upper bound at each step. The most important aspect of this approach to execution as compared to any previously considered in this chapter is that it operates *entirely* within the abstract domain. Because the abstract domain will typically aggregate together many (indeed, typically infinitely many) concrete states into single abstract states, the cost savings of performing analysis in the abstract domain can be tremendous.

This formula for generating an approximation to $\alpha_C(\mathbf{Fix}(I_C, S_0))$ admits to a particularly simple implementation as well. In particular, we begin with abstract state $s_0^{\#} = \alpha_C(S_0)$, and repeatedly apply $I_C^{\#}$ and take the least upper bound with the existing result until a fixed point is reached. Because $I_C^{\#}$ is monotonic, it can be shown that further applications of $I_C^{\#}$ will not raise the least upper bound. The result at the least fixed point is thus guaranteed to represent the full value $\sqcup (I_C^{\#})^n(s_0^{\#})$ over all n.

Our survey is nearly complete, but one final point must be addressed: While it has been shown that the value $\sqcup (I_C^{\#})^n(s_0^{\#})$ can be obtained by iterating $I_C^{\#}$ and $\sqcup$ until we reach the least fixed point, it has not been demonstrated that this convergence occurs within a finite time. This can be guaranteed if the

StateSequence$^{\#}$ domain is of finite height. The reason for this is simple: Because we are applying the $\sqcup$ operator for each iteration, the height of the least upper bound must increase within the domain by at least one for every iteration during which we have not reached a fixed point. If the domain is of finite height h, at most h iterations will be required before a fixed point is reached.)

In short, if we are willing to accept a calculated sacrifice of analysis precision, we can attain what are likely to be very large performance gains by conducting our analysis entirely in the abstract domain. By crafting our *StateSequence*$^{\#}$ domain so as to ensure that it is of finite height, we can further guarantee that the abstract interpretation will terminate regardless of whether or not the program being analyzed terminates. Moreover, we can perform this analysis so as to collect information regarding the behavior of the program on many different possible execution contexts (inputs) at once.

2.4 Summary

This chapter has examined execution in the standard, collecting, and abstract semantics, and used the latter as a model of program analysis. Execution was viewed from the perspective of "interpretation functions" which mapped states to states (or sequences of states to sequences of states). It was demonstrated that the formulation of an abstract collecting semantics could be used as a general methodology for performing program analyses in two ways.

First, if termination of the analysis is not required and if a premium is placed upon analysis accuracy relative to a small number of program inputs, it is possible to perform an analysis by successively taking the least upper bound of the abstraction of each state reached by the standard semantics interpreter I_S. This technique can incrementally but precisely build up the most accurate abstract model possible of the program's entire execution sequence, but can only do so for very few inputs at a time (typically just a single such input). Despite its cost and limitations on analysis breadth, this approach to analysis is used by many contemporary run-time analysis systems.

Secondly, if it is considered acceptable to sacrifice some analysis precision, it is possible to guarantee analysis termination using static analysis. This technique operates entirely within the abstract domain, and makes use of an "abstract interpreter" $I_C^{\#}$ that maps approximations to state sequences to approximations to state sequences. The analysis is carried out by repeatedly stepping $I_C^{\#}$ forward and accumulating the results of each iteration using the least upper bound operator. This process is continued until a fixed point is reached (which is guaranteed to occur in finite time if the (abstract) *StateSequence*$^{\#}$ domain is of finite height). This relaxation algorithm can be used to collect information regarding program behavior in response to arbitrarily many different inputs. The fixed point that is found

represents the results of the program analysis. This is the approach to abstract execution taken within this thesis.

	$\sqcup\alpha_C(I_S^n(\{s_0\}))$	$\sqcup(I_C^{\#})^n(\alpha_C(S_0))$
Informal Description	Run-Time Analysis	Static Analysis
Primary Analysis Domain	Concrete States	Abstract State Sequences
Analysis Precision	Exact.	Inexact.
Analysis Termination	If program terminates.	Guaranteed[6]
Primary Implementation Effort	Definition of α_C	Definition[7] of $I_C^{\#}$
Number of Inputs simulated.	One.	Many
Capacity for Modeling Uncertainty	None (all values concrete)	Arbitrary.

Table 1 : Comparison of the Run-Time and Compiled Analysis Approaches

[6] The termination is guaranteed if the StateSequence$^{\#}$ domain is of finite height.

[7] Note that this definition can take place either explicitly – by building an fully articulated abstract interpreter – or *implicitly*, by the construction of a compiler that produces $I_C^{\#}$ automatically, given the definition of the program to be analyzed. This approach is discussed at length in Chapter 7.

Chapter 3 Semantic Extensibility

3.1 Introduction

Program analyses have traditionally been hard-coded to collect certain types of information with regards to program operation. While this approach promotes greater analysis efficiency, it greatly limits the generality of the analysis tool. The design presented in this thesis takes quite a different tack. Rather than permitting the analysis to run on any program and achieving efficiency by specializing the analysis mechanisms to collecting particular analysis data, TACHYON allows the analysis to collect any of an unbounded variety of analysis data, and achieves efficiency and other benefits by specializing the analysis mechanisms to operate on a particular program.

The last chapter established a formal framework for understanding program analyses as abstract executions. This chapter first presents the conceptual basis for the *parameterization* of analysis to collect different varieties of analysis information. This section focuses in particular on demonstrating the safety of a broad range of semantic extensions to the fundamental analysis mechanisms.

Rather than making use of a simple parameterization scheme, however, this thesis advances a less restrictive framework that allows for *extensible* analyses. In particular, TACHYON maintains a user-defined "approximating" value and state domain with respect to which all program behavior is determined. The user then has an option of supplementing the normal information maintained on program values and state with custom user-defined information. This user-defined information can be calculated and accumulated during program analysis, but the resulting analysis must logically subsume the analysis that would be conducted in the absence of this information. (Slightly more formally, we can say that each interpretive "step" of the analyses conducted with the user-defined information approximate to a corresponding step within the analysis that would be conducted without that additional information.) By permitting the user to define custom analyses by dynamically "subclassing" existing analysis semantics, this approach reduces the amount of work required by the user to implement a new analysis technique, lessens the risk of implementation error, and permits an arbitrary number of such semantics to work in concert. Because this extensibility is implemented by means of delegation, it is can be performed at runtime, during the analysis operation.

Section 3.2 begins by discussing some of the broad motivations for semantic extensibility in program analysis. Section 3.4.1 begins by noting the shared mathematical framework underlying all forms of program analysis performed by means of abstract execution, regardless of which particular abstract semantics are involved. Section 3.5 then continues to build upon this framework to establish route to semantic *extensibility*. In particular, Section 3.5 characterizes two fundamental types of information manipulated during analysis, while Section 3.6.3.1 establishes a precise meaning to the notion of an "extended abstract domain". The section then applies this concept to the information carried by abstract states[8] and abstract values during analysis. This allows us to characterize what is meant by an "extended semantics analysis" in Section 3.6.1, and to establish a methodology for safely and dynamically "subclassing" and reusing existing analyses. While this methodology extends the user considerable freedom in selecting a custom semantics, the freedom is not without constraints. The final section discusses semantically extending program values that are drawn from the familiar 3-level domain. Values within this domain admit to particularly simple extensions both conceptually and in implementation.

Chapter 7 and Chapter 8 examine a technique for specializing the abstract interpretation-based analysis to the particular program being analyzed. The specialization yields a "compiled analysis" which is fixed to a particular program, but which maintains the component-based approach to semantic extensibility established in this chapter.

3.2 Motivations

There are a variety of motivations for permitting flexibility in the information collected by a program analysis algorithm. All of the major motivations seek to address broad user needs by empowering the analysis user and reducing or eliminating artificial or unhelpful restrictions on what aspects of analysis that user can access or customize. The most important of these goals are listed below:

Permitting Inspection of Analysis Rules. Traditionally, the rules by which program analysis has been conducted have been hidden out of the sight of the analysis user. While this has been motivated by the laudable desire to shield the user from needless complexity, when adhered to rigidly it can greatly complicate the interpretation of puzzling analysis results. Because analysis rules must operate in the absence of complete knowledge of program input, program flow, and programmer intent, the results returned by program analysis are necessity approximate. A fundamental question that frequently arises in

[8] To be precise, we should actually say that the extended semantics interpreter operates on extended abstract state *sequences*, for an abstract *collecting* semantics interpreter maps abstract state sequences to abstract state sequences. But it is convenient to think of the interpreter operating on extended semantics on account of the partitioning discussed below.

program development is whether an unexpected action or report from the programming system results from some inaccuracy in analysis, or reflects a genuinely problematic condition in the program. (For example, a compiler's report of a return from a function that is not associate with a proper return value could be an indication of a potential run-time error, but could also reflect an overly conservative estimate of possible paths of control flow.) Insistence on hiding analysis rules from the user can make the interpretation of such error messages very difficult.

For systems making use of run-time instrumentation, the rules for the instrumentation are hard-coded into the instrumentation software, and the transformations necessary for instrumentation have typically been performed directly to the user's object code or executable. A user desiring to understand the exact character of the transformations applied (for instance, to understand the evidence backing dubious report of a potential error) must rely upon assembly language dumps of the code or upon glimpses attained through a debugger.

Compilers or programming environments that perform program analysis at an earlier stage of the development process (e.g. to extract optimization-related or debugging-related information) typically offer the user even fewer options for understanding the analysis process. The analysis carried out by such systems is typically performed directly on an internal representation of the program by code deep within the bowels of the compiler or development environment. In many cases, the results of this analysis are discarded after being used internally for a specific purpose (e.g. optimization). Short of tracing through the execution of the analysis code, the analysis rules are not available for user inspection.

Customization of Information Collected. As will be seen in detail later in this chapter, analyses of program behavior frequently exhibit remarkable similarities in their overall structure and operation. It is therefore frequently straightforward to extend an analysis to collect additional information. For example, it would typically be a simple matter to modify an analysis that collects information on the line of allocation of pointer values to collect information specifying the time at which such values were created. As a second example, a compile-time analysis that implements the traditional optimization of copy propagation could easily be broadened to keep track of information regarding the constant values associated with program quantities as well. (Indeed, the algorithm to perform copy propagation in traditional compiler systems is also typically responsible for constant propagation). Providing the analysis user with the ability to extend the information collected by a particular analysis algorithm permits a simple means of extracting a variety of information from the program without the traditional need to create a custom analyses framework.

Flexibility in Analysis Heuristics. The modification of analysis rules is desirable not only for changing the types of analysis information collected, but also in refining or improving the accuracy of particular types of analysis. Certain types of analyses are implemented using heuristic rules that make assumptions concerning the way that programs manipulate particular varieties of information. Hard-coding analysis rules eliminates the opportunity to change or customize those heuristics in contexts where those assumptions are inaccurate and where other heuristics may be more accurate or effective. This inflexibility not only prevents the user from customizing the variety of analysis information collected; but can also undermine the accuracy of or benefit derived from the analysis.

For example, consider a compiler that analyzes a program to collect information for the sake of optimizing that program. For some imperative languages, an important part of that process is *alias analysis* – analysis of the pointer usage of the program to recognize cases in which a pointer could refer to the same data manipulated by a variable or other pointer. Typically, the presence of aliases requires the compiler to avoid performing certain optimizations in order to guarantee correctness. Unfortunately, alias analysis is frequently inaccurate. In most cases, this inaccuracy takes the form of being too conservative in its judgement of what program quantities can be aliased. This form of error can lead to needlessly distracting warning messages and/or the failure to take advantage of (sometimes-significant) opportunities for optimization.

In a small but especially problematic set of cases the heuristics used for alias analysis can be insufficiently conservative, and can fail to recognize possible aliasing behavior. (For example, insufficiently strong escape analysis can lead the analysis to overlook cases where routines outside the scope of analysis may modify stack-based or other data within the program being analyzed). The consequences of this form of analysis inaccuracy are much more severe, in that the results can trigger unsafe "optimizations" that actually change the semantics of the program and cause incorrect operation.

Program environments have traditionally provided the user with only coarse-grained control over the heuristics used during the analysis process. For example, users of an optimizing compiler typically have the option of disabling all or some optimizations when compiling a particular source code module. Some compilers may offer the user slightly more control by supporting pragmas or other compiler directives. Although such traditional mechanisms for controlling analysis serve an important practical purpose, such features limit the level of control that can be exercised by the user by conflating the analysis with the particular heuristics that are used to perform it. A more desirable solution would offer the user the additional option of modifying the heuristics used to implement the analysis, either for the program as a whole, for sections of code, or for particular program variables or quantities.

Avoiding Semantic Exclusivity. Traditional analysis tools have required separate analyses to collect disparate analysis information. A less restrictive strategy would permit "mixing and matching" of the analysis information collected, so that a particular analysis run could be used to simultaneously collect any number of different types of data on program operation.

Broadened Accessibility. While the standalone analysis applications common in today's programming tools market frequently provide rich presentations of the program analysis information they collect, typically the user is offered little or no access to this data. Program analysis systems such as profilers, optimizing compilers, API conformance checkers, and memory management tools typically discard the information collected once it has been internally processed. At best, such systems generally store the information in proprietary data formats for future use by the same system. Consequently, even in the many cases in which the information collected by such systems exactly suits user needs, the data is inaccessible for the purposes of manual or programmatic manipulation.

In a market where componentization has transformed almost all categories of software – from core office applications to middleware and back office systems – this approach seems increasingly anachronistic and restrictive. Applications in most popular categories of software expose rich language-independent hooks to internal functionality, and many provide internal scripting languages which provide programmatic control over the operation of the software and access to the data being manipulated. These changes have spawned a generation of "glueware" – productivity-enhancing custom software crafted from large prefabricated components.

The need for similar accessibility is particularly acute in the area of analysis tools due to the fragmented character of the market and the large number of sources of program analysis information. The potential for synergy among program development components spawned by scripting and shared access to data is tremendous: The code repository and versioning system, bug and anomaly tracking systems, automated test packages, source code editor, debugger, compiler, third party analysis tools, browser, and other common components of a programming environment all keep track of extensive amounts of closely related information. The sharing and control of such information by internal or external software offers tremendous potential for the development of powerful custom or shrink-wrap "glueware" applications, and has the opportunity to provide the programmer with considerably greater understanding of diverse aspects of program operation.

3.3 Implementation Issues

As is typical when designing a generic framework, we would like to make the semantic parameterization mechanisms as flexible as possible, while keeping a lid on the associated performance costs.

Traditionally, frameworks have had a choice between two types of customization, each with its own characteristic tradeoffs: *Static* and *Dynamic*.

Statically customizable systems allow a user to extend the system during development, either through subclassing, by means of type parameterization, or by adding source code to that provided by the framework. Examples of such systems are code generators such as YACC and LEX, windowing frameworks provided by MacApp or (to a lesser degree) MFC[9]. Static frameworks typically allow significant savings in development time and offer high performance, but lack the flexibility that is sometimes desired. Because such systems require the interfaces to the pre-built components to be defined during development, it is typically not possible to extend such frameworks during the execution of the program. The capacity to customize such systems therefore lies solely with the program *developer*, rather than with the *user* of such systems. There have been a number of past efforts to create generic *static* frameworks for conducting program analysis (see, for example, [Yi and Harrison 1993]). While such approaches have offered the prospect of faster and safer analysis development, they do not address many of the motivations for user customization discussed in Section 3.2.

By contrast, *dynamic* customizable frameworks allow for customizing the system in ways that were not anticipated when the program was originally designed. Modern techniques of dynamic customization frequently make strong use of *polymorphism*, in which the code can manipulate any implementation adhering to a certain abstract interface. Studies have demonstrated that although polymorphic manipulation of values does incur some additional performance overhead, the expense is not great.

In contrast to previous approaches, we are seeking a framework that would permit the user to create and use new semantics at run-time. We will thus make use of dynamic approaches to extensibility, and polymorphism will take a central role in the techniques proposed below.

[9] Note that MFC contains significant amounts of run-time extensibility, as evidenced by the reliance on maintaining dynamic class information.

3.4 Formulating a Generic Analysis Framework

3.4.1 Abstract Execution: A Shared Mathematical *Framework*

The fundamental prerequisite for the creation of a semantically extensible model of analysis is the capacity to use a single, unified model to describe analyses that collecting very different sorts of information regarding program behavior. Viewing analysis as an abstract execution allows us to do exactly this: Recall that given a collecting semantics interpreter I_C, we can define distinct abstract interpreters $I_C^{\#} = \alpha\, I_C\, \gamma$ for any pair of adjoints α and γ that form a Galois connection.[10] We can think of $I_C^{\#}$ as executing a "single step" of the execution in the abstract domain that serves as the domain of γ and the codomain of α, in a way that approximates (i.e. is compatible with) I_C. The process of abstract execution is identical for any abstraction function α and concretization function γ that form a Galois connection, or any abstract interpretation function $I_C^{\#}$ that observes the required correctness relations.

Recognition that there is a shared *mathematical* basis for very different types of analyses is a powerful impetus for a search for a common *implementation* framework for such analyses. In particular, to realize the goal of semantic extensibility, we would like to construct a "generic extensible abstract interpreter" $M_E^{\#}$ that is capable performing sound abstract execution under a wide range of abstract semantics (i.e. for a wide range of Galois pairs (α,γ)). For example, depending on the particular abstract domains employed, we would like the same extensible abstract interpreter to be capable of performing constant propagation, def-use analysis, collecting reference count information, generating compiled code.

While the existence of a shared mathematical framework is suggestive, the reification of a practical framework for analysis requires much more work. A particular mathematical mapping $I_C^{\#}$ can have an arbitrarily large number of possible implementations. Designing a generic interpreter $M_E^{\#}$ requires recognition and modeling of the broad similarities in possible implementations strategies for abstract interpreters, as well as the careful use of principles of engineering design. The next several sections examine these issues, and successively refine an approach to these issues.

3.4.2 Semantic Parameterization

A natural next step towards the construction of a common framework for abstract execution is to recognize the familiar functionality of the single step interpreter $I_C^{\#}$. In particular, the function of $I_C^{\#}$ is to map from an abstract state to a new abstract state according to the actions performed by the instruction

[10] For any non-trivial analysis in an imperative language, α and γ are likely to be rather complex, as they map between a set of states and an abstract state set. For instance, α must carry all of the information regarding an observed state sequence into the abstract state domain, and γ maps an abstract state set back into the set of all possible states that could have generated that state set.

(e.g. statement or declaration) at the current instruction point[11]. All abstract interpreters for a given language operate on the same set of constructs, and must adhere to the same semantic invariants in order to remain sound. Augmented with some additional constraints, these requirements can be used to shape the supported range of abstract interpretations to fit with what can be implemented in a common *parameterized* framework.

As a particular example as to how these constraints can lead to parameterization within a common framework, we consider the implementation of a single construct – the conditional. Figure 20 shows the semantics of a conditional statement. Figure 21 demonstrates the semantics of the conditional in the presence of abstract domains associated with potential uncertainty regarding the truth status of the predicate. Note in particular that the information present may be insufficient to determine which branch of the conditional will be executed. It is therefore sometimes necessary to approximate the result of *both* branches of the conditional. The result of this execution is obtained by taking the least upper bound of the state resulting from each branch.

$$C[\text{if } B \text{ then } C_1 \text{ else } C_2] = \lambda e.\lambda c.\lambda s.(B\ s) \rightarrow (C_1\ s)\ \square\ (C_2\ s)$$

Figure 20: Semantics of the If Statement in the Concrete (Standard) Semantics.

$$A[\text{if } B \text{ then } C_1 \text{ else } C_2] = \lambda e.\lambda c.\lambda s.(\text{cases } (B\ s) \text{ of } (\text{isTr}(b) \rightarrow (C_1\ s)\ \square\ (C_2\ s))\ \square\ \text{isUnit}(x) \rightarrow (C_1\ s)\ \sqcup\ (C_2\ s)$$

Figure 21: Semantics of the If Statement in an Abstract Semantics.

By virtue of the correctness requirements incumbent upon it, any abstract interpreter must handle a conditional in a manner that *approximates* that shown in Figure 21. This still leaves open a broad range of potential implementations for an abstract interpreter.

The fact that all abstract interpretations must maintain a common behavioral similarity with respect to constructs such as the conditional suggests the potential for representing them as parameterizations of a common set of mechanisms. In order to do this, however, it is necessary to somewhat restrict the

[11] Note that this glosses over the point that each abstract state may in general have *more* than one possible point of execution. (For example, evaluating a conditional's predicate that returns $\top$ will entail evaluation of *both* the consequent and the alternative of the conditional – but only a single abstract state. As a result, the abstract state must summarize both possible paths of execution. We allow for this by specifying only one of the possible subsequent points of execution as the "current" state, and the others as "pending" states that will be "awoken" at a later point (before the execution of the surrounding construct or subroutine). This adds further complexity to the adjoints α and γ, but would appear to be the best way to model the typical uncertainty regarding paths of execution that obtains in the abstract semantics.

supported set of interpretation mechanisms. In the case of the conditional, for example, we restrict ourselves to handling abstract interpretations which not only approximate, but must adhere *precisely* to the semantic rules specified in Figure 21.

Restricting the set of supported abstract interpretations in this way allows us to express the handling of the conditional construct in any abstract interpretations using the *same* semantics shown in Figure 21. Implicit in this code is an "overloading" of the $\sqcup$ operator for application to each different type of abstract state.

$$A[\text{if B then } C_1 \text{ else } C_2] = \lambda e.\lambda c.\lambda s.(\text{cases (B s) of (isTr(b)} \rightarrow (C_1\ s)\ \square\ (C_2\ s))\ \square\ \text{isUnit(x)} \rightarrow \text{LUB}($$
$$(C_1\ \text{GLBWithTruePred(s, b))}, (C_2\ \text{GLBWithTruePred(s, Not(b)))}$$

Figure 22: Semantics for Conditional in a Parameterized Semantics.

More explicitly, we can describe a "parameterized analysis" $I_p^{\#}(s, LUB, \ldots)$: $(State^{\#} \times (State^{\#} \times State^{\#} \rightarrow State^{\#}) \times \ldots)$ which explicitly takes as arguments functions performing the state manipulations appropriate for the desired abstract state (e.g. implementing the *LUB* function according to the structure of the semantic domain of interest.. Figure 22 shows a schematic representation of the conditional construct semantics for $I_p^{\#}$. With the specification of the appropriate abstract state and functionals, this interpretation function can be used to perform an "interpretive step" for any analysis that can be implemented using the conditional semantics shown in Figure 21.

Recognizing that the fundamental types of objects being manipulated in an analysis are abstract states and abstract values, we can use polymorphism to formulate an "object oriented" approach to semantic parameterization. In particular, we can implement methods such as *LUB*, *GLBWithTruePred* and *Not* in the code above as *method invocations* on abstract states and abstract values. The implementations of these methods for a particular semantic domain encode the semantic rules associated with that domain. Figure 23 shows a pseudocode example of the abstract execution of a conditional in an object-based parameterization framework. By virtue of the fact that the abstract state and abstract value objects encapsulate both their data (approximating or collecting information about their concrete semantics analogues) and the semantic rules for manipulating it, parameterization of this framework is particularly simple: All that is required is to "plug in" abstract values and abstract states matching the desired interfaces. The system described in this thesis makes use of this approach.

```
A[if B then C1 else C2] = λe.λc.λs.(cases (B s) of (isTr(b) → (C1.Execute(s)) ▯ (C2.Execute(s)))
▯ isUnit(x) → C1.Execute(s.GLBWithTruePred(b)).LUB(C2.Execute(s.GLBWithTruePred(b.Not())))
```

Figure 23: Semantics for Conditional in a Parameterized, Object-Based Semantics.

3.4.3 Semantic Restrictions

The shoehorning of abstract interpretation into a common parameterized framework does not come without its costs: Whether implicit or explicit, the choice of the mathematical parameterization framework upon which analysis is based invariably constricts the spectrum of analyses that can be expressed by the framework. In essence, by insisting upon a shared mathematical superstructure, we circumscribed the allowable behavior and ruled out the capacity to express analyses that fall outside of that range. To give a trivial example for the toy case of the conditional shown in Figure 21, the parameter semantics shown Figure 22 is not capable of describing an analysis which *always* approximates a conditional by the least upper bound of both paths. Similarly, the semantic characterization of Figure 22 cannot describe an analysis which maintains a monotonically increasing approximation to the results of analyzing the construct (such as is used in [Chen 1994]). While the semantic requirements discussed for the toy example above are hardly onerous, the collective cost of requiring adherence to a framework with many such rules can be substantial.

While we do not present a formal specification of the TACHYON parameterization framework in this section, the rather simple framework implicit in the interfaces specified in Chapter 5 and Chapter 6 and the compilation frameworks of Chapter 7 has very definite costs in terms of the generality of analysis. [Chen 1994] presents an efficient worklist algorithm whose implementation does not fall within this framework.

3.5 Semantic Partitioning

3.5.1 The Goal of Extensibility

The section above provided a strategy to achieve semantic *parameterization* within the abstract interpreter $I_C^{\#}$ by the use of polymorphism. This methodology allowed for the abstract value or abstract state semantics to be set by any implementation that observed the appropriate interfaces necessary to "plug in" to the analysis framework.

---

[12] As noted above, the encapsulation of many possible points of execution into a single abstract state further complicates the definitions of the upper and lower adjoints α and γ. But we are specifying $M_C^{\#}$ rather than the Galois connections themselves, and this need not serve as a practical roadblock. Were we to want to demonstrate the correctness of our abstract interpreter or prove that the Galois connection sound, this would require some care.

But this chapter is not merely about semantic *parameterization*, but about semantic *extensibility*. While parameterization lends great flexibility to the manner in which analysis is conducted, it omits an important part of the story. In particular, we use the term "semantic extensibility" to denote a very particular form of semantic parameterizability: To wit, a form of parameterizability in which the user has the option of *extending* (i.e. building upon rather than simply replacing) another semantics. This form of parameterizability has the virtues of greatly simplifying both the implementation of that semantics and the demonstration of the safety of the implementation. In essence, the user's semantics can "subclass" the base semantics, and thereby inherit all of the advantages of the base semantics while maintaining additional information and behavior. Appealing to this analogy, we will frequently use the term "derived semantics" interchangeably with that of "extended semantics", and will dub the semantics being extended the "base semantics."

Given the desire for an extensible semantics, what remains is to provide the formal understanding on which an implementation can be built. The first part of the section characterizes the two types of information being manipulated within an analysis, and the roles played by the information maintained in each domain. We then continue on to describe the decomposition of the analysis abstraction function introduced in Chapter 2 into two pieces that mirror this partitioning; this lends an understanding to how the partitioning relates to the analysis as a whole. Section 3.6.1 formally characterizes an "extended domain" and applies this abstraction to the particular case of the information manipulated during analysis and formulates a constraint to which extended semantics must adhere in order to remain safe approximations to the base semantics. With this constraint and formal model in mind, the next chapter turns to the engineering of a framework that implements the approach.

3.5.2 Formal Foundation

The previous chapter characterized two sorts of analyses processes.

- In *runtime analyses*, we compute $\sqcup\alpha_C(I_S^n(\{s_0\}))$ – in essence, running the program from state to state on a particular input, according to the interpretation function I_S^n. For each concrete state $I_S^n(\{s_0\})$ so reached, the analysis extracts the desired information and accumulates it using the lattice operator $\sqcup$. We noted that at each point in the analysis process, a runtime analysis system maintains two types of information – an approximation to the concrete state $I_S^n(\{s_0\})$, and the partially constructed information being collected.

- In *abstract execution*, the system performs analysis as the computation of a fixed point of the abstract collecting interpreter: $I_C^{\#}$. In particular, the analysis calculates $\sqcup(I_C^{\#})^n(\alpha_C(S_0))$. This is performed by mapping the original set of input states into the abstract domain, and then executing entirely in that domain.

It is a simple matter to re-express the computation that carries out an abstract execution to demonstrate the manipulation of the same two sorts of information seen during runtime analysis. In particular, we can consider a *partitioned* version of the abstract collecting interpreter: $I_P^{\#}$: $(\delta\times\omega)\rightarrow(\delta\times\omega)$, where ω represents the (partially constructed) information being collected about the program, and δ the approximation to the current abstract state.

Given an abstract standard (i.e. non-collecting) semantics interpreter $I_S^{\#}$, we define $I_P^{\#}$ as:

$$I_P^{\#}(\delta_0, \omega_0) = \{ I_S^{\#}(\delta_0), \omega_0 \sqcup \Omega(\omega_0, \delta_0) \}$$

Each step of the abstract execution thus collects some information on program behavior (in the first half element of the pair), and maintains an approximation to the current abstract state (in the second pair element). In each step of the abstract interpretation, Ω is used to extract the desired information to be collected from the current abstract state and combines it with previously collected information. Given this formulation of abstract execution, we can specify define the result ω_* of an analysis as the second element of a fixed point of $I_P^{\#}$. Slightly more formally,

Given the minimal n such that

$$(I_P^{\#})^n(\alpha_P(S_0), \perp) = (I_P^{\#})^{n+1}(\alpha_P(S_0), \perp)$$

We define

$$\omega_* = (I_P^{\#})^n(\alpha_P(S_0), \perp)\downarrow 2$$

This formulation of abstract execution suggests that we can collect different sorts of "ω" information concerning program behavior using the same abstract interpreter $I_S^{\#}$ by employing different functions Ω. Alternatively, we can contemplate collecting the same type of ω information while making use of δ information collected by different interpreters $I_S^{\#}$. Taken in combination with a *generic* analysis strategy (such as that presented in the previous section), we can imagine making changes to the ω and δ mechanisms transparently.

In the next section, we examine some of the intuitions behind these observations, and then generalize the framework slightly.

3.5.3 Basic Intuitions

The last section briefly provided the formal basis for conceptualizing abstract execution as manipulating two conceptually distinct types of information:

- ω(**"Collecting") Information:** The set of information concerning the history of program behavior that is ultimately desired. This is information that will be accumulated during analysis, and is typically retained for use following analysis. Presumably, some of this information is available at the beginning of computation. The remainder is built up according to *user-defined* rules that define how to combine and manipulate information within the ω domains for each statement and expression that is executed in the path of execution. While the information accumulated within the user domain during analysis can be fully as rich as desired, under no conditions can it constrain, restrict, or change the course of analysis.

- δ (**"Approximating") Information:** Approximations of the *current* abstract state that are necessary to conduct a sound and accurate analysis. This is information of transient – but crucial – character, used during analysis for the purposes of approximating that analysis in a sound manner. Frequently such information will go beyond what is strictly needed to guide a conservative analysis, and will include information designed to make the course of analysis more precise (e.g. to include fewer unreachable blocks or sides of conditionals that cannot in fact be executed). Improving the precision of this information adds to the precision of both the other δ information as well as the ω information. δ information frequently constitutes the large majority of the data manipulated during analysis, but it is ultimately thrown away.

While the information maintained for each of these purposes might sometimes overlap or be identical, the categories are conceptually distinct. Examples of clear partitioning of information abound. A few examples follow:

---

[13] While this might seem like a perverse choice to make, it could significantly reduce the cost of the analysis by substituting an inexact but highly efficient computation for a more precise but expensive calculation. For example, the rules for the ω-domains could accumulate history information that would help decide if a given analysis computation (e.g. detailed loop analysis) were worth performing with high precision. If it were decided that the cost outweighed the benefit, rules for the ω-domain could "override" the rules for the δ_{Statex} domain, assign $\top_{\delta}$ for appropriate portions of the state, and execution could continue following the loop.)

- In order to collect profile information from a program, the program must be completely executed. The accumulated analysis information (function entry counts, timestamps, etc.) represents the ω information; the δ information is the complete set of concrete program states encountered from start to finish of the program.

- Consider an analysis performed in order to deduce use-def [ASU] information for application to program splicing (this is the ω-information). In order to deduce such information in an accurate way, we may choose to maintain enough information concerning the values of program variables (the δ information) to avoid imputing uses to unreachable code.

- In analyzing the coverage of test cases, the program is executed with instrumentation to record which basic blocks have been encountered during execution. Here, the entire program state (δ information) is maintained in order to allow for the completely accurate collection of the boolean coverage record (ω information).

	δ Information	ω Information
Informal Name	Approximating Information	Collecting Information
Purpose	Approximating current state(s)	Providing information to user
Lifetime	Transient	Length of analysis
Control over analysis	Directs analysis	Epiphenomenal (can affect convergence time, but not expand or restrict set of possible run-time paths)
Object being approximated	Value or State	History of value or state
Loss of precision affects	Both δ and ω Information	Just ω information

Table 2: The Two Types of Analysis Information.

- In constant propagation of an applicative language, we try to discover cases where program quantities are associated with fixed constants (the ω information). In order to find this information accurately,

we must maintain extensive information regarding program closures (δ information). In another analysis, the reverse may be true, and information on the values of program variables in the current state (δ information) may be maintained to allow for improved quality closure analysis.

- Many optimizations of an imperative language collect and specialize based only on information regarding scalar program quantities (the ω information), but require detailed pointer modeling (associated with a great deal of δ information) to improve the precision of analysis.

Table 2 summarizes the characteristics of the two types of analysis information.

3.5.4 Information Manipulation in an Abstract Interpreter

As was briefly mentioned in the last section, both the abstract state and abstract value[14] can have components of both ω and the δ information. Like the per-state δ information manipulated by a run-time analysis system, the δ information within an abstract interpreter is short-lived. Within a standard semantics interpreter, each side-effect (e.g. a statement) mutates certain δ-information within program state, and each expression evaluation maps input δ-information to distinct output δ-information. The situation is similar within the abstract semantics: Expression evaluation leads to a constant computation of new (approximate) δ-information, and each side effect mutates the δ-information maintained within the state. There are some notable differences, however:

- **Non-Local Changes.** Within the standard-semantics, a particular state mutation is tightly localized in character, affecting only a very small set of locations in the state. By contrast, a given side effect within the abstract semantics can affect the analysis system's knowledge regarding a far broader set of possible program quantities. (For example, the abstract semantics emulation of a single assignment to an array element at an imprecisely bounded offset can require updating the δ-information for each element in the entire array.) This tends to make δ-information within the abstract semantics more ephemeral than its standard semantics counterpart.

- **Non-Linear Execution Path.** The standard-semantics interpreter operates along a single path of execution, with states following one another in a sequential fashion. Within the abstract semantics, however, the situation is more complicated: Imprecise knowledge of program runtime values within

[14] Note that while we speak here of the "abstract state" and "abstract value", it should be kept in mind that these are in fact abstractions of state *sequences* and value *sequences* – a point critical to the operation of the collecting domains.

the abstract semantics can lead to uncertainty regarding the paths of execution taken at runtime. To generate a sound approximation to any possible runtime path of execution, the system must approximate the results of executing any legal path of execution. Within this framework, there is no simple, linear ordering among abstract states encountered during execution – two arbitrary states may be drawn from two disjoint paths of execution. As will be seen Chapter 7, TACHYON deals with the possibility of multiple potential paths of execution abstract execution in a sequential manner. In particular, the system successively "suspends" all but one of the potential paths of execution, evaluating each path in turn from the point and state at which they split. This means that the δ-information within the state associated with a "pending" thread must be preserved until the thread is "re-awoken" for execution. In contrast to the previous difference cited, this can require a longer lifetime for δ-information than is common in the standard semantics.

Given an understanding of these differences, we note that δ-information is limited to the "active" states (the current state as well as any states associated with "pending" paths), while ω-semantics information is maintained for both the past and current states.

New ω- information is created in the process of computing each successive construct in the program, while mapping each state to the next. Some of this information may be maintained transiently, while in other cases it is squirreled away for use later in analysis, for post-processing, or for presentation to the user. While this "instrumented" information is collected during program operation, it is primarily epiphenomenal in relation to that analysis: We would not expect it to be directly used to influence or modify the course of program operation. We will formalize this expectation in the following section, in order to gain a clearer notion of what is meant by "extending" an analysis. Nonetheless, the presence of this information may significantly affect the running time of the analysis, by potentially extending the height of the abstract domain.

3.6 Combining Generic and Partitioned Abstract Execution

3.6.1 The Basis

The section above characterized abstract execution as manipulating two types of information, and provided some intuition as to the nature of this information. The definitions above formulated an abstract partitioning interpreter $I_P^{\#}$,

$$I_P^{\#}(\delta_0, \omega_0) = \{\ I_S^{\#}(\delta_0),\ \omega_0 \sqcup \Omega(\omega_0, \delta_0)\ \}$$

This interpreter maintains the approximating and collecting information in different compartments. Section 3.4 introduced the notion of a "generic" abstract interpreter, which can be parameterized to collect different types of information on program behavior by implementing the same set of semantic rules in different ways. We can combine the results of these two sections by building $I_P^{\#}$ in a generic execution framework.

This approach is motivated by the observation that the semantic rules necessary to collect information on program behavior can frequently be very naturally formulated as abstract operations on abstract data types (specifically, abstract states and abstract values). Thus, while the *functionality* of our system remains identical to that indicated by $I_P^{\#}$ above, the *implementation* of $\Omega(\omega_0, \delta_0)$ is as a set of semantic rules operating on abstract value and abstract state objects.

Taking advantage of the notion of a generic abstract interpreter, we can implement both the rules for manipulating ω and δ information as implementations of the generic rule templates associated with a generic abstract interpreter. The envisioned system thus segregates ω and δ information within each abstract value and abstract state, and maintains separate rules for manipulating ω and δ. These rules will vary according to the type of information to be collected and the character of the approximation to the current state that is desired. In effect abstract values and abstract states carry around information both with respect to their complete computation history (collected information), and with respect to the particular concrete values or states that they approximate.

3.6.2 Abstract Value and Abstract State Structure

We will now add some structure to the idea of maintaining ω- and δ- information within each abstract value and abstract state. We begin by noting that the ω- and δ- information components of both abstract values and abstract states are complete partial orders (cpos), associated with top and bottom elements and an ordering relation $\sqsubseteq$. We therefore have four cpos:

δ_{Value} with $(\top_{\delta\text{-}Value}, \bot_{\delta\text{-}Value}, \sqsubseteq_{\delta\text{-}Value})$.

δ_{State} with $(\top_{\delta\text{-}State}, \bot_{\delta\text{-}State}, \sqsubseteq_{\delta\text{-}State})$.

ω_{Value} with $(\top_{\omega\text{-}Value}, \bot_{\omega\text{-}Value}, \sqsubseteq_{\omega\text{-}Value})$.

ω_{State} with $(\top_{\omega\text{-}State}, \bot_{\omega\text{-}State}, \sqsubseteq_{\omega\text{-}State})$.

Given these complete partial order domains, we can further define an "extended value domain" E_{Value}= $\delta_{Value} \times \omega_{Value}$ and an "extended state domain" E_{State}= $\delta_{State} \times \omega_{State}$. Because each of the factors in the product is itself a complete partial order, the extended product domains will themselves be cpos. The extended value domain E_{Value} is a cpo with top element ($\top_{\delta\text{-Value}}$, $\top_{\omega\text{-Value}}$), bottom element ($\bot_{\delta\text{-Value}}$, $\bot_{\omega\text{-Value}}$), and an ordering relation $\sqsubseteq_{E\text{-Value}}$ defined as (a,b) $\sqsubseteq_{E\text{-Value}}$(c,d) $\Leftrightarrow$ a $\sqsubseteq_{\delta\text{-Value}}$ c & b$\sqsubseteq_{\omega\text{-Value}}$ d. Similarly, the extended state domain E_{State} is a cpo with top element ($\top_{\delta\text{-State}}$, $\top_{\omega\text{-State}}$), bottom element ($\bot_{\delta\text{-State}}$, $\bot_{\omega\text{-State}}$), and the ordering relation $\sqsubseteq_{E\text{-State}}$ defined as (a,b) $\sqsubseteq_{E\text{-State}}$(c,d) $\Leftrightarrow$ a $\sqsubseteq_{\delta\text{-State}}$ c & b $\sqsubseteq_{\omega\text{-State}}$ d. The height of the new extended value and extended state domains is simply the sum of the heights of the appropriate subdomains.[15] Thus, Height(E_{Value})=Height(δ_{Value}) + Height(ω_{Value}), and Height(E_{State})=Height(δ_{State}) + Height(ω_{State}).

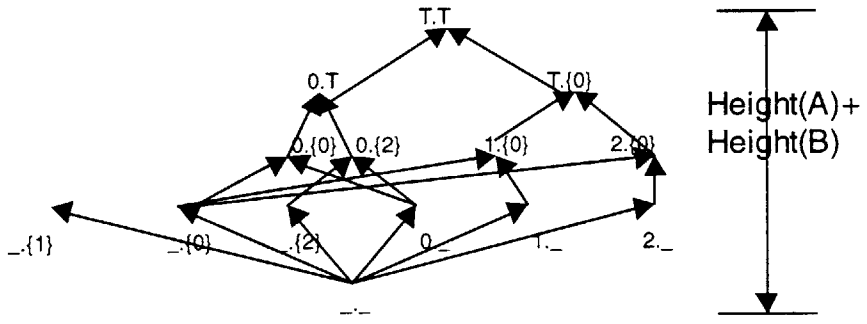

Figure 24: A Depiction of a Product Domain and its Longest Chain.

Following our establishment of E_{Value} and E_{State} as full-fledged complete partial orders, we will now declare them as the domains to be used for the abstract value and abstract state, respectively. That is, whenever we perform an abstract interpretation using the $I_P^{\#}$ outlined above, we will use an abstract value domain that is a product of the two (user-parameterizable) δ_{Value} and ω_{Value} domains, and an abstract state domain that is the product of the other two parameterizable domains δ_{State} and ω_{State}.[16] This formalization provides the desired extensibility alluded to in Section 3.5.2: By adhering to a particular "approximating"

[15] That the height of a product domain A × B of two cpos A,B is simply Height(A)+Height(B) can be easily verified by recalling that the height of a domain is the length of the *longest* chain in the domain, and by counting the steps necessary to reach ($\top_A$, $\top_B$) from ($\bot_A$, $\bot_B$).

[16] Note that while we spoke above of allowing the parameterization of the *entire* abstract value or abstract state domain as a unit, all parameterization within the extensible strategy takes place at the domain level Instead of parameterizing the behavior of the entire domain, we must individually parameterize the subcomponents of the domain.

value semantics δ_{Value} and independently "plugging in" an "collecting" semantics ω_{Value}, we have *extended* the operation of the δ_{Value} semantics itself to collect the additional user-defined semantics information. This is accomplished while preserving the same general flow of analysis. Plugging in an alternative collecting semantics ω_{Value}' creates another extended semantics resting on the same approximating value semantics δ_{Value}. Similar functionality is available in the state domain.

While it is gratifying to recognize the essential characteristics of extensibility within the formulation of the extended value and state semantics above, additional work remains to be done. In particular, we wish to allow not just extending the behavior of a particular isolated approximating domain δ (as achieved by adding a particular ω-domain), but extending the behavior of *any* existing value or state semantics (as specified by some combination of a particular approximating domain and an arbitrary number of collecting domains $\omega_1 \ldots \omega_n$). The next two sections tackle each of these issues in turn.

3.6.3 Extended Analyses

3.6.3.1 Handling Multiple Collecting domains

As defined above, the extended semantics for either a value or a state is specified as a product domain, with the approximating semantics domain δ in the first element of the pair, and a collecting domain ω in the second element. Although this is a convenient manner for representing the extended semantics, it is less flexible than we might ideally wish. In particular, rather than being restricted to performing analysis operations on only a single user-defined semantics, we would like to be able to offer the user the capacity to maintain an *arbitrary* number of user-defined abstract domains ω_i. This capability fosters modularity by allowing the user the option of adding an additional user-defined collecting domain to any existing analysis, and permits the collection of an arbitrary number of additional pieces of information for a particular analysis execution.

Recall that it is the approximating information δ that constitutes the approximation to the possible set of concrete values that can be taken on by a program value or state. Extending the ω information thus has no affect over the *precision* with which values are approximated. Instead, adding domains to the ω information augments the variety of information that can be maintained regarding a particular value or state's history or context.

Figure 26 illustrates an abstract domain that involves an arbitrary number of custom user domains. For the sake of simplicity, this figure assumes that the entire analysis takes place under a uniform extended semantics – that is, using a single unified abstract domain.

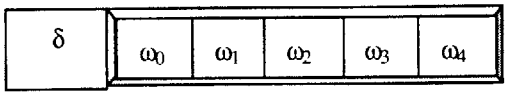

Figure 26: The Structure of the Extended Domains.

Typically we would expect that each domain within the ω information would be defined and evolve largely independently of all the other domains. (For example, a user-defined state domain collecting program slice information would probably evolve independently of one concerned with reference counting or recording suspect API calls). There may, however, be circumstances in which the semantic rules associated a given ω domain could make use of information associated with the another domain within the same value. To allow for this, we permit the rule for combining values of a certain ω domain to fetch the information associated with a particular domain tag within the same value.

Given $I_S^{\#}$: $\delta_{State} \rightarrow \delta_{State}$
Define $State^{\#} = \delta_{State} \times \omega_{State,0} \times \omega_{State,1} \times \ldots \times \omega_{State,n}$
Define $I_P^{\#}$: $State^{\#} \rightarrow State^{\#}$, such that
$I_P^{\#}(\delta_{State}, \omega_{State,0}, \omega_{State,1}, \ldots, \omega_{State,n}) = \{ I_S^{\#}(\delta_0), \omega_{State,0} \sqcup \Omega(\omega_{State,0}, \delta_0), \omega_{State,1} \sqcup \Omega(\omega_{State,1}, \delta_0), \omega_{State,2} \sqcup \Omega(\omega_{State,2}, \delta_0), \ldots \omega_{State,n} \sqcup \Omega(\omega_{State,n}, \delta_0) \}$
Execution begins with $I_P^{\#}(\alpha_P(S_0), \perp, \perp, \ldots, \perp)$

Figure 27: Definition of $I_P^{\#}$ in the Presence of Arbitrarily Many Collecting Semantics.

The extension of the ω information from a single domain to multiple domains is a natural one, and requires little additional mechanism. We might naturally anticipate that similar benefits would arise from disaggregating the δ information in a similar manner. Unfortunately, this is not the case: ω and δ information are qualitatively different in their use and manipulation within the analysis, and it is neither feasible nor desirable to equip δ information with the same dynamic extensibility that is used for ω information. The next section turns to examine this issue in more detail.

3.6.3.2 Motivations for a Single-Domain Approximating Semantics

The last section introduced a means to dynamically extend the ω information with an arbitrary number of subdomains. This increased the flexibility of the analysis, by permitting the user to collect different types of information concerning program operation at different times.

It is tempting to consider taking the same route here, by allowing dynamic extension of the approximating information. Unfortunately, while this adds flexibility to the analysis, it also gives rise to a great deal of complexity. There are two particularly egregious causes for this complexity: The cross-domain interaction and reasoning needed to empower operations to yield results with maximal precision, and the difficulty of combining information maintained in distinct domains into a form usable by the surrounding analysis machinery. Both of these complications prevent extensions from being made in a clean and modular manner to the δ portion of the state and value semantics. Instead, the constraints would necessitate that each such extension be cognizant of and potentially interact with each other subdomain of δ information – a requirement that greatly inhibits scalability in this area. The next two subsections briefly discuss each of these issues in turn.

3.6.3.2.1 Cross-Domain Reasoning

Recall that the δ information concerning a program value is distinguished from ω information in that it describes approximations to the concrete values that can occur at runtime. The aim in choosing to augmenting the δ (rather than ω) information of a value is to add to the *precision* with which that value is modeled within the analysis framework. Unfortunately, we cannot in general exploit this additional precision without the complexity that arises from reasoning about the information maintained in each δ domain. More particularly, while the result of a value operation will yield elements in all δ domains, computing the result in any one of those domains may require drawing information *simultaneously* from all δ domains. While the approximation allowed by reasoning about all δ domains will yield the most accurate approximations possible to value or state operations, it comes only at the cost of significant complexity.

To make the discussion more concrete, we will consider a particular case in which cross-domain reasoning is necessary. Consider an integer value containing two δ domains: An *EquivalenceClass* and an *ExtendedSign* domain. (The first of these domains is similar to the "Identity" attribute maintained in [Osgood 1993], and allows identification of two values that are known to be equal, even if the particular value they hold is unknown. The *ExtendedSign* domain records the approximation to whether a Value is known to be negative, non-positive, zero, non-negative, or positive and is a specialization of the "disjoint interval" approximating domain discussed in Chapter 11.). If we multiply or divide two values, computing the resulting *ExtendedSign* domain element will require information from both the *ExtendedSign* domain and from the *EquivalenceClass* domain (to find out whether the values being combined are the same.). If the values are associated with *known* sign, we can use such information alone to yield the maximally accurate response. But if those operands are classified as $\top$ in the sign domain,

we can still use information from the *EquivalenceClass* domain to recognize cases where two identical values are being combined, and yield a non-negative response for such cases.

Similar issues arise in handling combinations of many other potential δ domains. Allowing for inter-domain coordination is not terribly difficult, but it does require some framework for interaction between domains (the capacity to delegate a method to multiple domains and combine the responses into a single result for each δ domain). More importantly, the need for such coordination inhibits scalability, by requiring the code implementing a given domain to consider its interaction with and implications for the knowledge maintained in other domains.

3.6.3.2.2 Constraints from the Analysis Framework Interface

As discussed in previous sections, δ information is used during abstract execution to approximate run-time states and values so as to bound the possible paths of execution, the consequences of side-effects (e.g. array or pointer writes), the destinations of function calls, etc. In order to allow the analysis framework to perform effective bounding of this sort, the δ information for a value or state must frequently provide a *single*, unified approximation to that quantity. For example, the computation of a predicate must yield a single approximation to the truth status of the predicate result in order to determine which subset of conditional branches should be executed. To enable true dynamic extensibility of the δ information, any additional domains added to that information must be able to combine the knowledge maintained within those domains with the information maintained in all other domains to yield the single approximations required of program operations. No matter how or where synthesis takes place – in the state machinery or within the value operations themselves – the flexibility of the domain and the combinatorial character of the cross-domain interactions combine to make the integration very challenging.

3.6.3.2.3 Run-Time Extensibility

The section above noted the difficulties associated with handling the interactions among a large number of possible δ domains. While this is an onerous task in itself, it becomes more difficult when the δ information can be extended at run-time.

In particular, allowing run-time extensibility of the δ information eliminates any opportunity for *a priori* knowledge of the domains making up that information on the part of creator of a new domain. As a result, the creator is required to take into account the possibility that the domain being implemented will or will not coexist with *any* other possible δ domain, and provide the necessary reasoning to handle any *subset* of possible domains. Because the number of possible domain sets increases exponentially with the

number of domains, this is not a feasible strategy for situations in which there are a large number of possible δ domains.

While a specification of how multiple δ domains interact is important to empowering the additional precision that is sought when adding a new δ domain, it is not in general practical to specify how a new domain will interact with any possible combination of δ domains. The very desire to provide additional flexibility to the δ information by means of run-time extensibility thus inhibits the prospects of using such information in the most effective and beneficial way. The next section discusses an alternative approach to achieving extensibility in the δ information. This approach makes a calculated sacrifice of flexibility for the sake of simplifying the semantic extension process and allowing such extensions to make use of the maximum precision possible.

3.6.3.2.4 Approximating Semantics Extensibility

The past several sections have discussed difficulties associated with the prospective use of multiple domains for maintaining δ information. Although the capacity to extend the δ information with additional domains is desirable from the standpoint of flexibility and implementation reuse, it requires complex cross-domain reasoning, and appears very difficult in the context of run-time extensibility.

This requirement stands in contrast to the situation for ω information, which is not used directly during analysis and can be divided among multiple subdomains without need for coordination or inter-domain interaction. (While new ω information is always being created from combinations of values holding pre-existing ω information, the new values can typically be independently computed for each ω domain without reference to the information present in other ω domains.)

Given these constraints and conflicting tensions, this thesis adopts a compromise approach: The goal of achieving run-time extensibility of the δ domain is dropped. Only *static* extensibility is supported, which permits a given δ domain extension to be aware of the complete structure of the δ information at design time. Given the static extensibility of the approximating domains, there is no need to maintain multiple δ domains at run-time. In effect, all of the different δ domains can be shoehorned into a single implementation. TACHYON thus supports only a single δ domain, which can be statically extended by subclassing an existing implementation at the source code level.

While more restrictive than the approach taken for the ω domain, this methodology preserves the extensibility of the δ information (albeit in a less flexible form), and allows realistic achievement of the higher precision that is sought when extending this information

3.7 Conclusion

This thesis an *extensible* analyses framework that both allows for easy parameterization of the system with new domains and for straightforward "subclassing" of program behavior. The essential step in this approach is the division of abstract states and abstract values into two components:

- **Approximating Domain.** Each program object (abstract value or abstract state) has a single, distinguished guiding domain. This domain approximates the states and values present at run-time, but accumulates no information on program behavior to present to the user. By virtue of determining the set of run-time states that must be approximated by the analysis (in effect, guiding the analysis), the approximating domain influences the precision with which analysis is carried out for *all* domains.

- **Collecting Domains.** A given program object can be associated with an arbitrarily large number of user-defined "collecting" domains. In contrast to the approximating domain (which approximates the states and values present at run-time), each of these collecting domains approximates state and value *history*. As such, they serve to collect user-specified information on program behavior. The addition of a collecting domain to an existing set of domains in effect "subclasses" the analysis to accumulate the additional set of information. While approximating domains directly determine the set of concrete states approximated by the program, the collecting domains are *epiphenomenal*, merely observing – and not influencing – the course of analysis.

Having established the foundation for the generic analysis framework, the following chapters continue on to specify the means by which the framework is *implemented*. The next chapter begins this process by sketching the overall architecture of the analysis.

SECTION II: ENGINEERING

Chapter 4 TACHYON Architectural Overview

This chapter provides a brief overview of TACHYON architecture, and discusses an aspect of the extensibility implementation that is common to both abstract states and abstract values. Subsequent chapters will describe the design and engineering of each component of the architecture in detail.

4.1 Implementing Extensibility via Interface-Based Polymorphism

The sections above characterized the mathematical basis for extensibility. Within this framework, an analysis can be expressed using a functional that expresses rule templates applied to elements drawn from the appropriate sequence of abstract domain elements. The parameterization of each such domain is achieved by passing functions to this functional. These passed functions encapsulate the semantic rules by which these domains evolve. Section 3.4.2 provided a glimpse of the mathematical basis for parameterizing a particular domain. This framework may be helpful from a conceptual point of view, and offers a certain amount of guidance in *implementing* the domains in an analysis framework.

The implementation of TACHYON mirrors the mathematics in making use of a common framework consisting of abstract values and states subdivided into domains, as shown in Chapter 3. Rather than using a functional to accomplish the semantic parameterization of each such domain, the system makes use of *interface polymorphism.* Each abstract state or value contains a single δ-domain and an arbitrary number of user-defined ω-domains. A particular step of the analysis (for example, the evaluation of an integer "+" expression or an allocation expression) is associated with a particular method in either the abstract value or abstract state domains. In order to evaluate this step, the run-time system invokes that method on the appropriate abstract value or abstract state domains. Figure 28 schematically illustrates the coupling between the abstract analysis engine and the domain implementations.

For each domain, the coding of this method encapsulates the semantic rule associated with that sort of operation for that particular domain. For example, when presented with an integer "+" operation, a value domain collecting point-of-origin information might union together the points of origin associated with the operands to the expression, while a value domain associated with the code generation process might generate code to perform the operation. Similarly, a state operation (such as a "split" resulting from an unknown predicate of a conditional, or a least upper bounds operation to some label) will be associated with corresponding calls to state-specific methods.

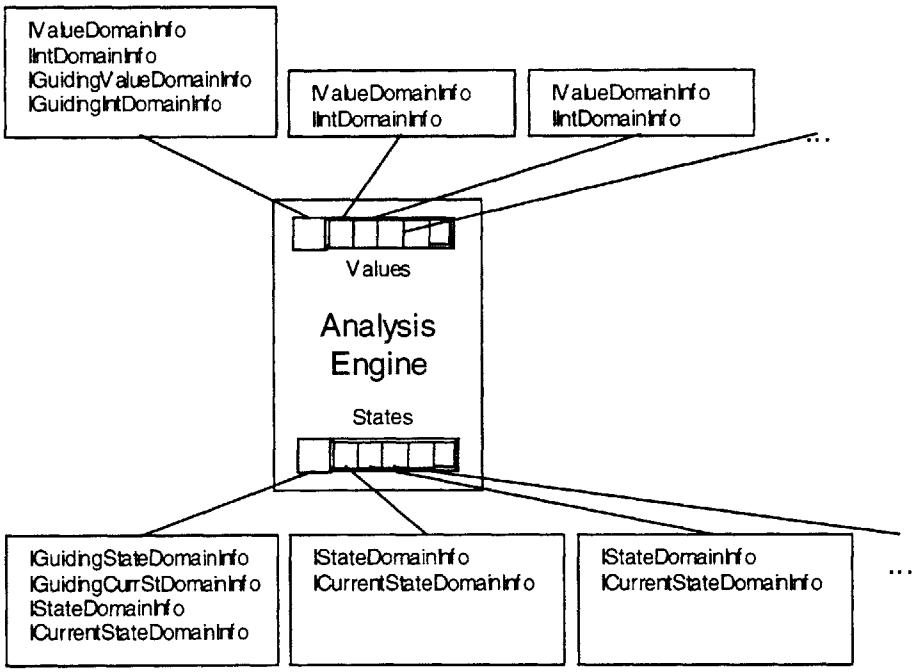

Figure 28: Semantically Extensible Analysis with "Pluggable" User-Defined Domains.

While the quantum of parameterizability and extensibility is the *domain*, it is convenient to implement abstract states and abstract values as encapsulated objects created and manipulated by the flow of execution of the program. A particular abstract value or abstract state object will contain methods precisely mirroring those which can be applied to the domains, but which operate at the abstract value or abstract state level of abstraction. The abstract state and abstract value simply represents a convenient wrapper which desugars such methods into calls to the appropriate domain-level methods, and constructs new abstract value or state wrappers around the result. The operation of the abstract state and values does contain some subtleties, however, and will be discussed further in Chapter 6 and Chapter 5.

The design of natural value and state interfaces required care and experimentation, and those interfaces and some of the motivations shaping them are discussed in detail within subsequent chapters. The next chapter describes the interfaces to which implementations of value domains must adhere in order to "plug" into the abstract execution framework. Chapter 6 describes the more complicated abstract state domain interfaces in an analogous manner.

4.2 Creation of the Analysis Executable

Thus far, little has been said as to how the analysis mechanisms are implemented. Most abstract execution systems created thus far have been based on an abstract *interpreter*, which can operate over any user-specified program. It is certainly possible for the analysis described above to operate in this manner, making use of a general-purpose, extensible, abstract interpreter parameterized by both the program on which to operate and the domains under which the abstract interpretation will be carried out. For performance and other reasons, however, TACHYON opted for the "compiled analysis" methodology shown in Figure 29. Within this methodology, a compiler is used to translate each source code module in the program to be analyzed to its abstract semantics analogue, which can then be compiled and linked using a standard programming environment. The resulting "analysis executable" is specifically optimized to conduct analysis on the user's program, according to the semantic domains specified by the user.

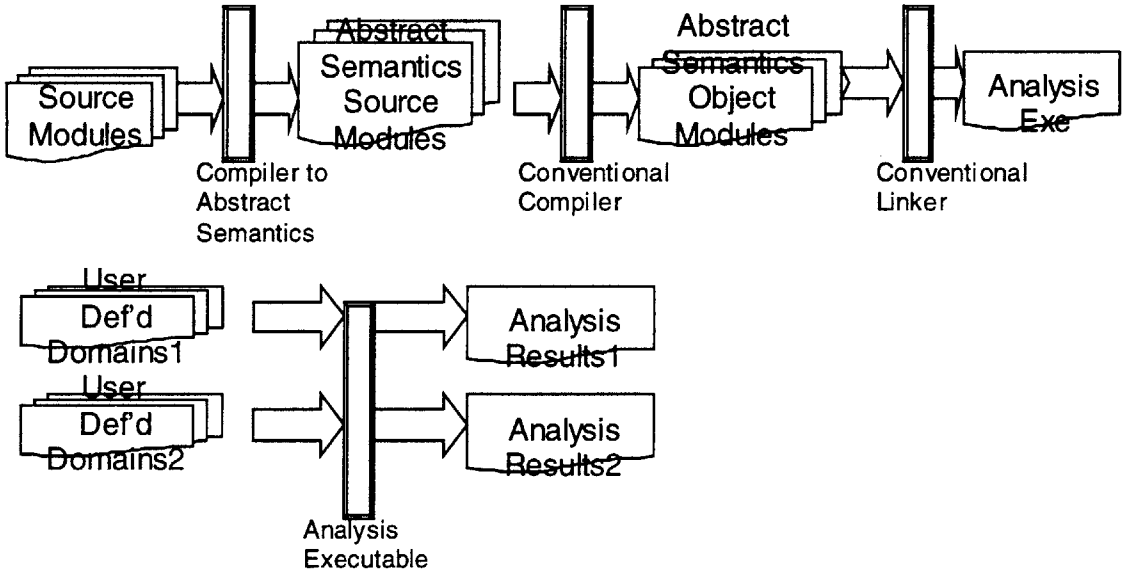

Figure 29: The TACHYON Approach to Abstract Execution.

Chapter 7 examines the interpretation/compilation tradeoff (including a discussion of a probabilistic model of the relative performance of compiled and interpreted analysis), and surveys issues related to the modeling of values and states in TACHYON's. Chapter 8 describes in considerable detail the rules used to translate a program into its abstract semantic analogue. The specification of these rules additionally helps to illuminate the design decisions associated with the domain interfaces presented in Chapter 5 and Chapter 6.

4.3 Implementation of the Value-Level Interfaces

As discussed in Chapter 3, TACHYON achieves extensibility by conceptually partitioning the abstract states and abstract values into a single approximating and multiple collecting subdomains. This section examines the *implementation* side of that approach, sketching the structure of the domain sequence and briefly discussing the means by which value-level operations are translated into calls to the domain-level interfaces.

4.3.1 Structure of the Abstract Values and States

Abstract values and abstract states look identical from the point of view of gross morphology. These objects differ only in that the sequence of domains in the abstract state must implement *state* domain interfaces, while those composing values must implement *value* domain interfaces. Abstract value and abstract state objects are implemented as an ordered sequence of domain elements, each associated with a specific user-specified domain. The first such domain element within each value is the distinguished user-defined *approximating domain*, which serves as an approximation to the run-time status of the modeled values and states in the standard semantics. This domain is associated with a zero-valued domain identifier, and must be present in every abstract value or abstract state object. The placement of the approximating domain in the first position in the sequence guarantees fast access to the domain in the not uncommon cases in which that domain must be treated specially (e.g. in testing the truth of a predicate).

The domains following the approximating semantics domain are *collecting* semantics domains and are sorted by their (positive integral) domain identifier. There can be arbitrarily many collecting domains in any given abstract value or state.

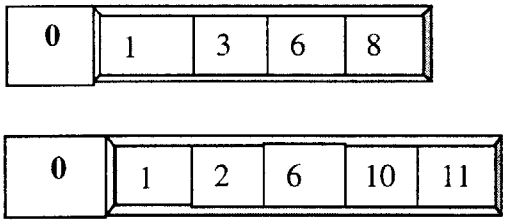

Figure 30: Example of Two Values/States From the Same Analysis containing Elements Drawn from Different Domains.

4.3.2 Domain Sequence Flexibility

TACHYON currently supports abstract values and states of the same type composed of *different* sequences of collecting domains. The capacity to deal with heterogeneous domain sequences allows the user greater flexibility during analysis. The two most notable aspects of this flexibility are:

- **Reduced Implementation Effort.** If the same set of domains were required for all states and values, the user would be required to implement a complete set of different types for each domain that was to be implemented (i.e. integer, double and pointer value domains, and a state domain).[17] Relaxing the assumption of a canonical domain sequence may allow the user to sidestep the need to implementation certain types of domains. For example, if the user implements the abstract integer value domain but not an abstract pointer value domain, subtraction of two pointers may yield an abstract value domain element that lacks the user-specified domain. The machinery currently implemented in TACHYON (partly shown in Figure 31) is sufficiently robust to allow such a value to combine with other values that do contain the user's domain without difficulty, by treating the absent element as a $\top$ value.

- **Capacity to Introduce Domains During Analysis.** By tolerating values containing different sequences of domains, the system allows a user to introduce domains into analysis values throughout the analysis process. This late binding of the domain structure allows the user to adjust the character of the analysis while the analysis is in operation. It is not expected that this capability would see widespread use. But the ability to dynamically add domains may be desirable in certain specialized circumstances in which extensive user interaction takes place, with changing needs throughout – for example, in adding additional debugging-oriented or visualization domains while trying to understand program behavior.

This section has briefly surveyed the structure of the abstract domains and the motivations for the loose restrictions on the abstract domain sequence. While these conventions allow for greater flexibility, they do come with a significant performance cost. This issue is discussed further in the next section, which examines the mechanisms used to translate from calls at the value level to invocations at the domain-interface level in more detail.

[17] The full set of types is required so that the system will be able to rely upon a value having certain domains, regardless of whether it was created by the same or a different domain.

4.3.3 Mapping Value- and State-Level Operations to Domain-Level Operations

The analysis code interfaces with abstract value and abstract state components at interfaces defined on that level. By contrast, the user parameterization of abstract value and state behavior takes place on the *domain* level. A compact but important piece of system behavior relates to the translation of abstract state and abstract value-level methods down to the domain level. This section examines issues associated with that translation. The examples are drawn from the code implementing the abstract value component, but isomorphic code is found in the abstract state implementation.

Figure 31 shows the core code that is responsible for combining two values using an operator. Because two values (or states) may or may not share the same domain sequence structure, domain identifiers are used to identify domain elements belonging to corresponding domains. (See Chapter 4.) The code examines the domain structure of the two values in order to determine the domain structure of the result, and then applies the corresponding domain-level operator to each pair of matched domains within the two operands. The output value is guaranteed to contain the union of the domains present in each of the operands, but calculating the pairwise domain combinations in the presence of differing domain structure requires some care. If a particular domain is associated with an entry in one operand but not another, the system models the other operand as containing an $\top$ entry for that domain.

While the flexibility afforded by a late-specified domain structure can be valuable, its cost is substantial. The running time and software engineering complexity of the operator is significantly increased by the fact that the system must tolerate combinations of values containing different sequences of value domains. Elided from Figure 31 in the interests of space is a piece of code performing a pre-scan of the operand domain structure. Nevertheless, the code does demonstrate the interpretive, data-dependent nature of the value combination. By contrast, a fixed domain structure would allow for a much more efficient, vector-style combination of domain elements. This algorithm would involve only a single pass over the algorithm rather than two, and no need for shaping control flow according to value data. In retrospect, it seems doubtful to the author whether the additional flexibility arising from late-bound domain structure is worth the cost. Fortunately, changing the system to a more straightforward and efficient vector-based combination of values would entail changing only a handful of routines, such as the one shown in Figure 31. This work essentially involves *simplifying* existing code, and thus would be would require exceptionally little effort.

```
void  ApplyBinaryOperatorToEachSemanticDomainPair(_IAbstractValue arg2, ProcessDomainSpecificVa
luePairs combiner)
  {
      int iArg1;
      int iArg2;
```

```
    if ((this.m_rgDomainSpecificValues[0].DomainId() != 0) || (arg2.Domain((short) 0).DomainI
d() != 0))

    throw new IllegalStateException("Missing approximating semantics at the beginning of the
set of semantic domains within an AbstractIntValue");

    int ctSemanticEltsArg1 = this.m_rgDomainSpecificValues.length;
    long domainIdArg1 = this.m_rgDomainSpecificValues[0].DomainId();
    int ctSemanticEltsArg2 = arg2.CtDomains();
    long domainIdArg2 = arg2.Domain(0).DomainId();
    int  ctDistinctDomains = 0;

    iArg1 = iArg2 = 0;
      …Code to determine the total number of domains in the resulting value

    // all the rest of the domains are disjoint.   (Note that in reality, only one of the
    //    arguments will have remaining domains at this point, so only one of the terms
    //    below on the rhs of the += will be non-zero.)
    ctDistinctDomains += (ctSemanticEltsArg1 - iArg1) + (ctSemanticEltsArg2 - iArg2);

    iArg1 = iArg2 = 0;
    combiner.SetCtDistinctDomains(ctDistinctDomains);
    _IValueDomainInfo eltArg1 = m_rgDomainSpecificValues[0];
    domainIdArg1 = eltArg1.DomainId();

    _IValueDomainInfo eltArg2 = arg2.Domain(0);
    domainIdArg2 = eltArg2.DomainId();

    while (iArg1 < ctSemanticEltsArg1 && iArg2 < ctSemanticEltsArg2)
    {
        if (domainIdArg1 < domainIdArg2)
        {

    if (combiner.ProcessDomainSpecificValuePairAndReturnFComplete(eltArg1, eltArg1.TopElt()))
          // ok, if we can combine the elements, do so
            {
                combiner.SetComplete();
                return;
            }
            iArg1++;
            if (iArg1 >= ctSemanticEltsArg1)
                break;
            eltArg1 = m_rgDomainSpecificValues[iArg1];
            domainIdArg1 = eltArg1.DomainId();
        }
        else if (domainIdArg2 < domainIdArg1)
        {

    if (combiner.ProcessDomainSpecificValuePairAndReturnFComplete(eltArg2.TopElt(), eltArg2))
          // ok, if we can combine the elements, do so
            {
                combiner.SetComplete();
                return;
            }
            iArg2++;
            if (iArg2 >= ctSemanticEltsArg2)
                break;
            eltArg2 = arg2.Domain(iArg2);
            domainIdArg2 = eltArg2.DomainId();
        }
        else
        {
            if (combiner.ProcessDomainSpecificValuePairAndReturnFComplete(eltArg1, eltArg2))
            {     // ok, if we can combine the elements, do so
                combiner.SetComplete();
                return;
            }
            iArg1++;
            iArg2++;
            if (iArg1 >= ctSemanticEltsArg1 || iArg2 >= ctSemanticEltsArg2)
                break;
```

```
            eltArg1 = m_rgDomainSpecificValues[iArg1];
            eltArg2 = arg2.Domain(iArg2);
            domainIdArg1 = eltArg1.DomainId();
            domainIdArg2 = eltArg2.DomainId();
        }
    }

    while (iArg1 < ctSemanticEltsArg1)
    {

    // by not having this at the END of the loop, we avoid an extra check of whether iArg1 is
out of bounds
        eltArg1 = m_rgDomainSpecificValues[iArg1];
        // ****this could be optimized for the LUB case, but never mind

    if (combiner.ProcessDomainSpecificValuePairAndReturnFComplete(eltArg1, eltArg1.TopElt()))
          // ok, if we can combine the elements, do so
        {
            combiner.SetComplete();
            return;
        }
        iArg1++;
    }

    while (iArg2 < ctSemanticEltsArg2)
    {

    // by not having this at the END of the loop, we avoid an extra check of whether iArg1 is
out of bounds
        eltArg2 = arg2.Domain(iArg2);

    if (combiner.ProcessDomainSpecificValuePairAndReturnFComplete(eltArg2.TopElt(), eltArg2))
          // ok, if we can combine the elements, do so
        {
            combiner.SetComplete();
            return;
        }
        iArg2++;
    }

    combiner.SetComplete();
    return;
}
```

Figure 31: Code to Map a Binary Value Operation into Corresponding Operations on the Semantic Domains.

4.4 The Toy Language

The past several chapters have characterized abstract execution in general terms, without restricting the discussion to a particular language. This section briefly describes the toy language handled by the TACHYON compiler and run-time. In the interest of making the presentation more accessible, we have shied away from a formal specification in favor of a more informal presentation. Readers interested in a more precise characterization of the language are referred to the grammar associated with the TACHYON parser.

Like [Osgood 1993], TACHYON implements only a small basis set of "Simple C" statements and expressions. The "desugaring" source-to-source transformer described in [Osgood 1993] can be used to transform most of the C programming language into the "Simple C" subset supported here. Small

extensions to the existing transformer or language would allow for *syntactic* support for the entire C language. Full support for the abstract interpretation of C would, however, require addressing some challenging issues, such as abstractly modeling untyped regions, abstract values read and written as different types, and unrestricted casting.

Table 3 shows the statement types currently supported by the TACHYON compiler and runtime. The supported functionality includes an iterative looping primitive, a conditional statement, statements to terminate or continue a loop, and a statement that returns a value. Note that an important control-flow operator not shown in the table is the function call. Function calls are currently supported by TACHYON, although operator-specific run-time support is lacking and fixed-point handling of recursive operators is designed but not yet implemented (see Chapter 8). While effective analysis of pointers-to-functions forms one of the central motivations for strong analysis, such pointers are not yet modeled in the TACHYON runtime, and calls through pointers to functions are not yet supported.

Statement	Notes

`while (predicate)`
`  stmt s`
`  statement`	

`if (predicate)`
`  stmt consequent`
`[else`
`  stmt alternate]`
	Else clause is optional

`Continue`	Continues nearest enclosing loop
`Break`	Terminates execution of nearest enclosing loop
`Return[(expression)]`	Terminates function execution. For non-void function, the result of *expression* is returned as the value of the function invocation

Table 3: Set of Statement Types supported by TACHYON

The language supported by TACHYON currently supports only two categories of values: Integers and Pointers. As in the C programming language, boolean values are modeled as integers. Currently, a single abstract pointer domain type models pointer values of all types; full support of C typing would require a more flexible modeling approach. (See Section 5.4.5.)

Table 4 shows the integer value operators supported by TACHYON. This set of operators models virtually all of the semantics of integer manipulation in C.

Category	Value Action	Notes
Integer Binary Operators	*	Multiplication
	/	Division
	+	Addition
	-	Subtraction
	%	Modulo
	&	Bitwise And
	|	Bitwise Or
	^	Bitwise Exclusive Or
	>>	Right bitshift
	<<	Left bitshift
Boolean Binary Operators	&&	Logical And (No short-circuit)
	||	Logical Or (No short-circuit)
Cross-type Binary Operators	==	Equality
	!=	Inequality
	<	Less than
	<=	Less than or equal to
	>	Greater than
	>=	Greater than or equal to
Assignment	=	
Unary Operators	~	Bitwise not
	!	Logical not
	-	Negative

Table 4: Integer Operators Supported by TACHYON

Table 5 shows the pointer operators supported by TACHYON. The TACHYON compiler for the most part adheres to a subset of the ANSI C pointer functionality. An important exception is that the system requires that allocation functionality be *statically* recognizable in the program text through a call to the allocation primitive *new*, rather than embedded in a call to inaccessible library routines (e.g. *malloc*, *calloc*, etc). Ability to recognize such primitives is critical for effective fixed-pointing of loops in the presence of allocation. These rules are very similar to those applied in C++. For the sake of canonicalization, all array indexing is translated into corresponding pointer manipulation. The "address of" operator is currently syntactically supported, but the compiler does not yet properly model escaped variables so as to allow for their access through pointers, although such support would require very little additional work.

Category	Value Action	Notes
Pointer arithmetic	+ (Pointer offset by integer)	Argument is an abstract integer
	- (Pointer difference)	Result is an abstract integer
Assignment	=	
Comparison	==	Result is an abstract boolean
	!=	Result is an abstract boolean
	<	Result is an abstract boolean
	=<	Result is an abstract boolean
	>	Result is an abstract boolean
	>=	Result is an abstract boolean
Creation	Call to allocation primitive *new*	Takes as argument the source location and a memory region allocated by the analysis machinery. Note that the memory region allocation is performed with explicit knowledge as to whether a fixed-point is being pursued.
	& (Address of variable operator)	Passed the memory region and offset of the variable whose address is being taken by the analysis machinery. This also takes an argument specifying source location, in order to allow the semantic domains to collect information on the source point of origin of pointer values. Not yet fully supported by compiler.
Dereference	*	Mapping depends on whether specifying lvalue.

Table 5: Pointer Operators Supported by TACHYON

4.5 Conclusion

Having briefly surveyed the overall architecture of the analysis system, we turn now to more detailed discussions of the interfaces through which abstract states and abstract values and are controlled by the analysis machinery.

Chapter 5 Interfaces for the Generic Value Domains

5.1 Introduction

Having briefly characterized the framework in which abstract execution takes place, and sketched how interface polymorphism could be used to construct a semantically extensible analysis system. This chapter focuses on the interfaces to which TACHYON requires that abstract value domains adhere for use within the analysis system. While this chapter will shed some light on the concerns and reasoning that shaped these interfaces, the description will remain incomplete unless it is considered in conjunction with the descriptions in the subsequent three chapters (describing the state interfaces and the compilation system).

One final note is in order. While implementation of the interfaces described in this chapter allows for participation of a domain in abstract execution, it guarantees neither correctness nor realistic running times for the abstraction. The *correctness* of a user's domain is guaranteed as long as the appropriate invariants are maintained, as described in Chapter 2 and Chapter 3. In particular, the abstract value history resulting from performing each abstract operation on a domain element must soundly approximate any member of the set of results of applying the standard semantics analogue of that operation to each possible value history combination represented by the operands. Beyond the correctness issues, the user must remain mindful of the running time implications of a domain as well. A domain could adhere to the interfaces presented here but have such a deleterious effect on the asymptotic or effective running time as to be unrealistic. Section 7.4 of Chapter 7 discusses the performance implications of the value domain height in more detail, Section 12.3 presents a approximating value domain for pointers that increases the asymptotic order of the analysis.

5.2 Value Taxonomy

To simplify the implementation process, only three scalar value types were modeled in the TACHYON system: Integers[18], double floating-point values, and pointers. As can be seen in Table 6 and Table 7, the collection of abstract value domain interfaces contains a similar division, but includes finer subdivisions as well.

[18] Recall that Integers are overloaded to describe both boolean and integral values.

Domain Name	Description
IValueDomainInfo	Common value domain interface to which value domains of any type must adhere. Provides access to common characteristics of all value domains, such as the domain Id, distinguished members of the domain, a text representation of the domain element, and the associated approximating domain.
IIntDomainInfo	Type-specific domain interface to which integer (including boolean) domains adhere. Includes method declarations for all common integer and boolean operations (both unary and binary), to implement $\sqcup$ operation (for both the case where a fixed-height lattice is required and when it is not), and conceptually class-level methods to create an instance of this domain from a manifest constant or for an uninitialized value. Conversion operators are supplied to convert integers to double values.
IDoubleDomainInfo	Type-specific domain interface for double floating point values. Methods are similar to those in the *IIntDomainInfo* interface, except that they do not include boolean operators or functionality. In addition, double values must support an arbitrary unary mapping function as well as conversion to integer values.
IPtrDomainInfo	The type-specific domain interface for pointer values. Includes boolean operations for comparing pointers, taking the distance between pointers, indexing off of pointers, both varieties of $\sqcup$ of pointers, and to create pointer values from allocations, uninitialized values, and from taking the address of values.

Table 6: General Value Domains.

Table 6 shows the interfaces required for implementation by all value domains – whether approximating (δ) or collecting (ω) in character. Table 7 illustrates the interfaces specific to *approximating value* domains. Such domains are called upon to not only respond to standard methods but also contain methods relating to their responsibility to direct the course of analysis observed by all domains. For example, boolean approximating value domains are responsible for indicating whether or not they are known to be true or false, integer approximating value domains must be capable of specifying the particular concrete value to which they correspond, and pointer approximating value domains have the responsibility of dealing with writes and reads of elements containing *all* domains to and from memory.

Table 8 illustrates which interfaces are required for implementation by each of the four varieties of value domains currently implemented. It should be stressed that while the selection of methods required for implementation by each of these varieties of value domains is far from arbitrary, the division of those methods between different interfaces is often extends beyond engineering and into the realm of

convenience and aesthetics. For example, some of the methods contained in type-specific interfaces (e.g. *IIntDomainInfo*) could reasonably be placed instead in the *IDomainInfo* (indeed, some of the methods in *IIntDomainInfo* partially or entirely duplicate functionality found in *IDomainInfo*). The primary considerations in each of these categories are as follows:

- **Aesthetics**. Capturing commonality among different interfaces in higher-level interfaces makes for cleaner designs.

- **Software engineering**. Hoisting methods to higher levels of abstraction and avoiding their duplication in other interfaces can help reduce maintenance effort. In addition, coding for routines located in high-level interfaces is simplified if one can avoid calling routines in less general interfaces. A countervailing pressure arises from the fact that it is frequently convenient to have narrowed versions of high-level routines defined in conceptually more specific interfaces, to avoid littering the code with narrowing casts.

Domain Name	Description
IGuidingValueDomainInfo	General interface implemented by all approximating domains, whatever the type of the run-time value they approximate. This includes methods for testing if the run-time value approximated by this domain is known to be a particular concrete value, and to retrieve the value returned by the current method in the approximating domain during the current operation.
IGuidingIntDomainInfo	Interface implemented by abstract value domains approximating integer or boolean values. Methods within this interface permit accessing whether or not the value is known to be true (non-zero) or false (zero) for booleans, and for retrieving a known concrete value and the maximum and minimum possible runtime concrete values for integers.
IGuidingDoubleDomainInfo	Interface implemented by any domain that approximates run-time doubles. Similar to the guiding integer domain, but lacks boolean operators.
IGuidingPtrDomainInfo	This interface describes the methods that must be implemented by any approximating pointer value domain. The interface contains methods to read and write each type of integral/boolean or pointer value.

Table 7: Interfaces Implemented by Approximating Domains.

- **Performance**. Because the interfaces are defined independently (rather than being derived from one another), both conceptual widening and narrowing operations are associated with significant performance costs. It is desirable to avoid such performance costs by defining related routines in the same interfaces. Note that this consideration places incentives on the duplication or near-duplication of methods in more specific interfaces rather than in the more general interface.

	Approximating	**Non-Approximating**
Integer/Boolean	*IValueDomainInfo* *IIntDomainInfo* *IGuidingValueDomainInfo* *IGuidingIntDomainInfo*	*IValueDomainInfo* *IIntDomainInfo*
Double	*IValueDomainInfo* *IDoubleDomainInfo* *IGuidingValueDomainInfo* *IGuidingDoubleDomainInfo*	*IValueDomainInfo* *IDoubleDomainInfo*
Pointer	*IValueDomainInfo* *IPtrDomainInfo* *IGuidingValueDomainInfo* *IGuidingPtrDomainInfo*	*IValueDomainInfo* *IPtrDomainInfo*

Table 8. Interfaces Required for Six Different Types of Value Domains.

5.3 Domain-Level Interfaces

Having presented an overview of the value domains and the context in which each is used, we turn now to examine each of the interfaces in detail. For each interface, the methods of the interface will be listed, annotated with notes on each. Additional comments will follow in the text.

5.3.1 General Domain Interfaces

The first set of domain-level interfaces to be discussed are those supported by *all* value domains. As noted in Table 8, there are four such interfaces, each of which is allocated a section below.

5.3.1.1 IValueDomainInfo

As noted in Table 6, the *IValueDomainInfo* interface presents the highest level value interface, and is the sole interface implemented by all varieties of value domains (regardless of the underlying type and whether they represent approximating domains or not). Several of these methods (e.g. *SetContext*, *FIsTopElt*) are called from the internal value combination machinery rather than from the top-level analysis process. The *DomainId* method is of particularly high importance and frequent use, as it allows us to determine whether two pieces of domain information held in two different values correspond to the same or different domains. It is only if they belong to the *same* domain that they combined.

Method	Description
IDomainInfo TopElt()	Primary use is as proxy for domain info when explicit info on this domain is absent, and for representing the information associated with uninitialized values.
IDomainInfo BottomElt()	Used to initialize values that accumulate information through application of the ⊔ operator.
int DomainId()	Tag used to identify type of domain info, and for pairing up information belonging to identical domains.
String Unparse()	For debugging and user convenience, all values are required to allow for textual representation.
boolean FIsTopElt()	Allows shortcuts in evaluation of domain operations.
boolean FIsBottomElt()	Allows shortcuts in evaluation of domain operations.
IAbstractValue Context()	Permits (indirect) access to the other domains contained within the surrounding abstract value. This is useful when accessing approximating domain information
Void SetContext(IAbstractValue v)	Typically called only by the internals of the framework, to establish the surrounding value.

Table 9: Methods in Interface *IValueDomainInfo*.

For the most part, the domain-level and value-level interfaces are disjoint and need not refer to one another; values are typically only combined with values, and the domains they contain with other domains. An important exception to this occurs in the *SetContext* and *Context* methods, which are used for setting and obtaining information concerning the value in which a particular domain is located. The *SetContext* routine is called by the value machinery to inform the domain about its surrounding value. The *Context* domain will typically be called by user code as an intermediate step in obtaining the approximating domain for the surrounding value. For example, the code in Figure 32 is excerpted from the integral code generation value domain, and handles binary combination operators. The routine first checks to see if the approximating domain being returned for the operation is precisely known at analysis time. If so, there is no need to perform the calculation at run time, and the code generated need only be a load of the appropriate value into a register. Otherwise, we call off to generate the code to perform the value combination, using as arguments the code to calculate each of the operands (i.e. the code generation value domain operands to the current method).

```
    private _IIntDomainInfo LocalBinaryCombineOrShortcut(_IIntDomainInfo v1, _IIntDomainInfo v2
, int opType)
    {

    _IGuidingIntDomainInfo intDomainMostRecentReturnFromGuidingDomain = (_IGuidingIntDomainInfo
) this.Context().GuidingDomainGuidingInterface().GuidingValueIntReturnValue();


    if (((_IGuidingValueDomainInfo) intDomainMostRecentReturnFromGuidingDomain).FConcrete())
        {

    // ok if we know that the result of this operation is a concrete value, just generate the c
ode to load it


    return (_IIntDomainInfo) this.CreateImmediateLoadValueOfSameType(intDomainMostRecentReturnF
romGuidingDomain.SSValue());
        }
        else

    return(( _IIntDomainInfo) this.BinaryCombine((SCGValue) v1, (SCGValue) v2, opType));
    }
```

Figure 32: Cross-Domain Accessing in a Sample Approximating Value Domain Implementation.

5.3.1.2 IIntDomainInfo

	Method	**Description**
Integral Binary	IIntDomainInfo operatorAdd(IIntDomainInfo r)	These operators encapsulate the semantic rules associated with each type of binary integral value expression.
	IIntDomainInfo operatorMul(IIntDomainInfo r)	
	IIntDomainInfo operatorSub(IIntDomainInfo r)	
	IIntDomainInfo operatorDiv(IIntDomainInfo r)	
	IIntDomainInfo operatorMod(IIntDomainInfo r)	
	IIntDomainInfo operatorArithAnd(IIntDomainInfo r)	
	IIntDomainInfo operatorArithXor(IIntDomainInfo r)	

	IIntDomainInfo operatorArithOr(IIntDomainInfo r)	
	IIntDomainInfo operatorRightShift(IIntDomainInfo r)	
	IIntDomainInfo operatorLeftShift(IIntDomainInfo r)	
	IIntDomainInfo operatorEQ(IIntDomainInfo r)	
	IIntDomainInfo operatorNE(IIntDomainInfo r)	
	IIntDomainInfo operatorLT(IIntDomainInfo r)	
	IIntDomainInfo operatorLE(IIntDomainInfo r)	
	IIntDomainInfo operatorGT(IIntDomainInfo r)	
	IIntDomainInfo operatorGE(IIntDomainInfo r)	
Int Unary	IIntDomainInfo operatorArithNot()	Semantic rules for unary integral expressions.
	IIntDomainInfo operatorNeg()	
Boolean Operators	IIntDomainInfo operatorLogicalAnd(IIntDomainInfo r)	The semantic rules for binary and unary boolean expressions (recall that this domain represents both boolean and integral values)
	IIntDomainInfo operatorLogicalOr(IIntDomainInfo r)	
	IIntDomainInfo operatorLogicalNot()	

Least Upper Bound routines.	IIntDomainInfo FiniteHeightLUB(IIntDomainInfo r, boolean[] fChanged)	This is used to perform a ⊔ in such a manner that the analysis must converge within the application of a finite number of such operators. This operator plays a critical role in the analysis of program loops (in which the analysis is responsible for terminating within a finite number of loop iterations, even if the loop itself is non-terminating). In such cases, it is used in the joins associated with backedges in the execution.
	IIntDomainInfo LUB(IIntDomainInfo r)	Performs a ⊔ operator where no convergence constraint is present. This operator is commonly used to implement non-backedge joins in the flow graph.
Miscellaneous and Convenience	Boolean FIsApproximatedBy(IIntDomainInfo arg)	
	_IDoubleDomainInfo operatorConvertToDouble()	Converts integer to corresponding abstract double domain.
	boolean FImmutable()	
	int DomainId()	For convenience, a duplicate of the *DomainId* operator of IValueDomainInfo.
	IGuidingIntDomainInfo AssociatedGuidingDomainInfo()	Provides narrowed, direct access to the approximating domain information associated with the value.
	IAbstractIntValue NarrowedContext()	A narrowed version of the *Context* routine.
Domain Info Creation	IIntDomainInfo CreateFromConstant(int v, int srcLoc)	Conceptually a class-level method. Called by the framework to create appropriate information for this domain when a constant value is encountered in the program text. Note that the "srcLoc" parameter provides the location of the constant in the program source, in order to allow the analysis to carry such information in values. The "v" parameter specifies the value of the constant being created. (A particular domain may discard one or both of these pieces of information.)
	IIntDomainInfo CreateUninitialized(int srcLoc)	Conceptually a class-level method. This is called by the framework when execution reaches a point at which an uninitialized value is created (e.g. an uninitialized variable or an uninitialized member of an array).

Table 10. Methods in Interface *IIntDomainInfo*.

The *IIntDomainInfo* interface constitutes the central interface for the implementation of integral and boolean domains, and contains most of the functionality used in working with such values. As can be seen from Table 10, the interface contains a large number of methods, roughly subdivided into four families.

The domain creation methods perform the essential function of creating new instances of this domain from manifest constants encountered in the source code and in cases where lvalues are not initialized.[19] Conceptually, these methods are class-level operations, but in the absence of a uniquely appropriate way of creating polymorphic class-level operations, we have chosen the convention of making such operations standard methods and invoking them on singleton prototypes of the domain. Note also that the creation routines accept as an argument a specification of the source code location at which the creation is occurring. This information allows the value domain to carry and manipulate such information in subsequent value combination machinery, and makes straightforward the collection of information about the origin of circulating program values.

```
_IAbstractIntValue i = intPrototype.CreateUninitialized(4);
```

Figure 33: Invoking the Domain Element Creation Routines through a Prototype Singleton.

The most frequently used and most numerous methods in the interface above are the domain information combination routines, which are responsible for carrying out the semantic rules for this domain for each value combination operator. Thus, we have methods corresponding to each of the operators that can be used to combine run-time integral or boolean values – abstract semantics analogues to "+", "-", "/", logical or and logical and, etc. Typically the implementation of such operators will draw information from one or more operands, process it in some way, and return the result of this processing. It is worth recalling that because we are working in an abstract *collecting* semantics, abstract values are in truth abstractions of value *history* (that is, of the entire process by which a value has been derived.) Thus, the handlers associated with such operators may collect and manipulate extensive information concerning the history of a value.

[19] Note that while uninitialized lvalues are always associated with the top element of the *approximating domain*, this is by no means the case for all value domains. For example, a value domain which kept track of whether particular values were guaranteed to rely only on initialized values would have a ⊤ element corresponding to "either initialized or uninitialized", and an uninitialized value would generate an element from the value domain associated with the label "uninitialized."

At a syntactic level, the least upper bound methods may resemble other value combination routines, but the semantics of the two types of routines are quite different. The methods associated with value combination routines are used for instances in which the user's code has made use of the corresponding standard semantics operator. In accordance with the rules of abstract interpretation specified in Chapter 2, the behavior of such methods must soundly approximate the behavior of the corresponding standard semantics operator. By contrast, the least upper bound operators are not abstractions of any standard semantics operators – they are simply operators defined on the abstract value domain. The return value of either least upper bound operator must represent a valid approximation to either of the arguments to that operator, but the two least upper bound operators differ in other ways. The *FiniteHeightLUB* operator must guarantee convergence to a fixed value after a finite number of applications. This operator is also responsible for returning a boolean value indicating whether the return value is still changing. (Note that because of the inability of the Java programming language to express call-by-reference or multiple return values, the boolean return value is actually passed as an array. The boolean return value indicates whether the result of the least upper bound is higher in the lattice than the value on which the operator is invoked.) By contrast, the LUB operator simply takes the least upper bound of two values in cases where no convergence criterion is required.

The convenience methods seen in the table above largely duplicate similar routines in the more generic *IValueDomainInfo* interface. One exception to this is the conversion operator *operatorConvertToDouble*. An important issue that bears emphasizing is that the presence of conversion operators among the methods that must be implemented by integral and double domains has implications for the manner in which such domains are specified. In particular, it requires that every double domain have an analogous integral domain, and vice-versa. In effect, it requires double and integral value domains to be simultaneously defined.

	Method	**Description**
Combination Routines	IPtrDomainInfo operatorOffset(IIntDomainInfo r)	Expresses the result of indexing off of the pointer by a specified number of bytes. Note that the parameter here implements IIntDomainInfo, but must correspond to the same domain (i.e. must have the same domain id). In the familiar terms of the C programming language, this describes adding an integer to a (character) pointer.
	IIntDomainInfo operatorSub(IPtrDomainInfo r)	Implements the semantic rule associated with subtracting two pointers.

	IIntDomainInfo operatorEQ(IPtrDomainInfo r)	This and the subsequent methods handle boolean comparison of pointers. Note that the results here are integral, but belong to the same abstract domain (have the same domain id).
	IIntDomainInfo operatorNE(IPtrDomainInfo r)	Same as above.
	IIntDomainInfo operatorLT(IPtrDomainInfo r)	Same as above.
	IIntDomainInfo operatorLE(IPtrDomainInfo r)	Same as above.
	IIntDomainInfo operatorGT(IPtrDomainInfo r)	Same as above.
	IIntDomainInfo operatorGE(IPtrDomainInfo r)	Same as above.
Least Upper Bound	IPtrDomainInfo FiniteHeightLUB(IPtrDomainInfo r, boolean[] fChanged)	Analogous to the same operation for the integral value domain – this method is used to take the least upper bound of pointers in cases where convergence is guaranteed after only a finite number of such operations. (See description in int value domain.)
	IPtrDomainInfo LUB(IPtrDomainInfo r)	Takes the least upper bound in cases where no convergence constraints are present. (See description in int value domain.)
Miscellaneous and Convenience	Boolean FisApproximatedBy(IPtrDomainInfo arg)	
	boolean FImmutable()	
	int DomainId()	Provides convenient access to domain identifier in this interface (duplicates functionality of method in IValueDomainInfo)
	IGuidingPtrDomainInfo AssociatedGuidingDomainInfo()	
	IAbstractPtrValue NarrowedContext()	

Creation Routines	IPtrDomainInfo CreateFromAllocation(IMemoryRegion region, int srcLoc)	Creates a pointer value from an allocation request. Note that the framework provides the pointer value with memory region information as well as the location in the source code of where the allocation takes place.
	IPtrDomainInfo CreateFromAddressOfVariable(IMemoryRegion region, int iOffset, int srcLoc)	Creates a pointer value from the address of a variable.
	IPtrDomainInfo CreateUninitialized(int srcLoc)	Creates an uninitialied pointer value (e.g. for an uninitialized variable or uninitialized member of an array).

Table 11. Methods in Interface *IPtrDomainInfo*.

5.3.1.3 IPtrDomainInfo

Table 11 shows the methods associated with the *IPtrDomainInfo* interface, which encapsulates the functionality specific to pointer value domains. Because the number of pointer operators in the standard semantics is rather less than the number of integral and boolean operators, the interface is smaller than that for the integral and boolean domains, although still of substantial size. Like the integral/boolean domain interface, however, the pointer value domain interface is divided into four families of methods.

Creation methods. As is the case for the integral/boolean domains, the creation methods are conceptually class-level but are defined as polymorphic methods invoked on prototype singletons. The three creation methods reflect the three ways in which pointer values can be created during execution of a program. In particular, pointer values can be created by taking the address of a variable, from an allocation request, and from an uninitialized lvalue (in a variable or allocated region of pointers). It is important to stress that the methods to handle the first two of these situations (*CreateFromAllocation* and *CreateFromAddressOfVariable*) are not themselves responsible for computing the location resulting from the allocation. Instead, the location is computed by the analysis machinery via a call to the abstract state domain and is specified via arguments to these methods. An example of this is shown in Figure 34. As was the case for the integral/boolean value domain, all of these methods take an additional argument that specifies the source location at which the value was created.

```
p = statePrototype.CurrentStateCurrentInterface().operatorAssignmentToPtrVariableFilter(53,p,st
atePrototype.CurrentStateCurrentInterface().AllocateIntArray(tempId0,fPerformingFixedPoint,5),5
);
```

Figure 34: Code Associated with Allocating a Memory Location.

Combination Routines. In contrast to the integral/boolean domain interfaces shown in Table 10, the interfaces for the pointer domain offer relatively few "inductive" routines that combine preexisting domain elements. Those that exist are divided into two groups.

The *operatorOffset* and *operatorSub* methods deal with computing the pointers from offsets or vice-versa. These routines differ from those seen in the *IIntDomainInfo* interface in that they bring together domain elements from the same domain (i.e. having the same *DomainId*) but different types. This implies that any domain id for which a pointer domain exists to have a corresponding integral domain having an identical domain id. As we will see in the next chapter, similar requirements apply in the context of the state domain.

The comparison routines (e.g. *operatorEQ*, *operatorLT*) compare pairs of pointers. Like the *operatorSub* routines, they require the return of an integral domain element with the same domain id as the current pointer domain element.

It is worth noting the conspicuous absence of one important class of combination routines within the domain above: Methods associated with rules for handling pointer reads and writes do not occur within the interface above. As discussed in Section 5.3.2.3 below, because unit of the storage in the abstract state is the *value*, the value-level methods handling the reading and writing of values are centralized in the *approximating* pointer domain interface (*IGuidingPtrDomainInfo*). Collecting domains may choose to "filter" their corresponding domain elements after they are read or before they are written, but the desire to maintain a centralized point at which such filtering could take place led to the placement of such routines in the *state* domain interfaces. Thus, any semantic rules that seek to collect information concerning memory access activity should be placed in the state interfaces (described further in Chapter 6.)

5.3.1.4 IDoubleDomainInfo

Table 12 shows the methods associated with the *IDoubleDomainInfo* interface that is required for implementation by any value that approximates or collects information on a double precision value at runtime. The interface is similar to that for integral values, except that it does not support boolean and bitwise operators and includes a few additional combination operators as well.

	Method	Description
Arithmetic Operators	IDoubleDomainInfo operatorAdd(IDoubleDomainInfo r)	
	IDoubleDomainInfo operatorMul(IDoubleDomainInfo r)	
	IDoubleDomainInfo operatorExp(IDoubleDomainInfo r)	In this domain only
	IDoubleDomainInfo operatorSub(IDoubleDomainInfo r)	
	IDoubleDomainInfo operatorDiv(IDoubleDomainInfo r)	
	IDoubleDomainInfo operatorNeg()	
	IDoubleDomainInfo operatorMap(IDoubleMap fn)	Performs arbitrary mapping
Conversion	IIntDomainInfo operatorConvertToInt()	Convert to integral analogue
Comparison	IIntDomainInfo operatorEQ(IDoubleDomainInfo r)	
	IIntDomainInfo operatorNE(IDoubleDomainInfo r)	
	IIntDomainInfo operatorLT(IDoubleDomainInfo r)	
	IIntDomainInfo operatorLE(IDoubleDomainInfo r)	
	IIntDomainInfo operatorGT(IDoubleDomainInfo r)	
	IIntDomainInfo operatorGE(IDoubleDomainInfo r)	
Lattice Operators	IDoubleDomainInfo FiniteHeightLUB(IDoubleDomainInfo r, boolean[] fChanged)	
	IDoubleDomainInfo LUB(IDoubleDomainInfo r)	
	IDoubleDomainInfo GLBWithTruePredicate(IIntDomainInfo vPredicate)	
Misc.	boolean FIsApproximatedBy(IDoubleDomainInfo arg)	
	boolean FImmutable()	
	int DomainId()	
	IGuidingDoubleDomainInfo AssociatedGuidingDomainInfo()	
	IAbstractDoubleValue NarrowedContext()	
Creation	IDoubleDomainInfo CreateFromConstant(double v, int srcLoc)	
	IDoubleDomainInfo CreateUninitialized(int srcLoc)	

Table 12. Methods in Interface *IDoubleDomainInfo*

Of particular note in this regard is the presence of an exponentiation operator (*operatorExp*) and a general-purpose unary mapping operator (*unaryMap*). The unary mapping operator can be used to implement general-purpose table functions (e.g. in system dynamics simulations such as that discussed in the Introduction), and to enforce predicates (e.g. as applied to the sample space domain of Chapter 11). This function makes use of mappings that adhere to the interface *IDoubleMap*, which is shown in Table 13. While the most widely used method in this interface is the *Map* method that performs a unary map

from an input double to an output double, the other methods play important roles in bounding the effects of the function on imprecisely known approximations to runtime values.

Method	Description
double Map(double d)	
double RangeMin()	
double RangeMax()	
double RangeMinForDomainInterval(double lower, double upper)	
double RangeMaxForDomainInterval(double lower, double upper)	

Table 13: The *IDoubleMap* Interface.

5.3.1.5 Summary

This section has summarized the structure and use of the general value domain interfaces that must be implemented by user-defined semantic domains. Most of the domain methods encode semantic rules to be fired in different contexts during analysis. Some of the most important of such methods dictate how new domain elements are initially created (e.g. when the address of a variable is taken or when a manifest constant or uninitialized variable occurs in the program text), and how such values are combined. Other interface methods for such domains take the least upper bound of two domain elements either for a fixed-point operation or for a simple join without convergence requirements.

5.3.2 Interfaces for Approximating Domains

Having described the type-generic and type-specific interfaces implemented by *all* value domains, we turn now to those implemented only by a very specific domain: The distinguished approximating domain. This domain accompanies every abstract value and serves to approximate the standard semantics values taken on by that abstract value at runtime. As shown in Table 7, approximating domains are responsible for implementing the type-generic interface *IGuidingValueDomainInfo*, as well as at least one of either *IGuidingIntDomainInfo* or *IAbstractIntValue*.

5.3.2.1 *IGuidingValueDomainInfo*

	Method	Description
Epistemic Categorization	Boolean FConcrete()	Indicates whether the value held by the approximating domain is precisely known. (i.e. Indicates whether the run-time value associated with this quantity is definitively known to be a certain standard semantics value.) Note that for *pointers*, this implies that the referent of the pointer is precisely known, and can be attributed to a single allocation occurrence.
Return Value Information	IIntDomainInfo GuidingValueIntReturnValue()	Returns the most recent abstract integral or boolean value returned by a method call to this domain. This permits other domains to "peek" at what was returned by the approximating domain and to shape their value accordingly.
	IPtrDomainInfo GuidingValuePtrReturnValue()	As above, but returns the most recent pointer returned from this domain.
	Boolean GuidingValueBooleanReturnValue()	Similar to the above, but returns the most recent boolean (not *abstract* boolean) returned from this domain. Note in particular that this includes booleans conceptually returned by FFixedPointLUB (but actually passed back through singleton arrays passed as arguments).

Table 14. Methods in Interface *IGuidingValueDomainInfo*.

The other method within the domain (*FConcrete*) is used to judge whether the identity[20] of the current value is precisely known.

5.3.2.2 *IGuidingIntDomainInfo*

The *IGuidingValueDomainInfo* interface serves as the type-generic approximating domain interface for values, and is of particularly simple composition. The majority of its methods exist to report the most recent return values of different types that have been returned by method calls to this domain element. The motivation for such methods is to allow collecting domains to easily shape their own return values to accord with those returned by the approximating domain. For example, a domain involved in performing

[20] Note that in the sense being used here, the identity of a value may be known without the precise standard-semantics value being known. For example, we may be able to uniquely distinguish a pointer value from all other pointer values, but not know its precise numerical value. This definition is largely one of convenience, and should probably be considered in greater depth. (For example, in certain contexts it may be desirable to know even greater information concerning pointers, such as the location of their referents relative to those areas referenced by other pointers.)

constant propagation would wish to know whether the approximating domain has returned a concrete value for a particular operation, and to annotate this fact. While implementing such a domain does not *require* access to the current return value[21], such access greatly simplifies the design of the domain.

	Method	Description
Boolean Operations	boolean FKnownTrue()	Indicates whether the run-time value approximated by this domain (when interpreted as a boolean) is known to have the concrete value "true".
	boolean FKnownFalse()	Same as above, but indicates whether the run-time value approximated by this domain is known to be false. Note that while it can never be the case that FKnownTrue and FKnownFalse obtain for a given value, all other combinations are possible.
Integral Operations	int SSValue()	If the run-time value approximated by this approximating domain is concrete, returns the standard semantics value represented by this domain. Otherwise, the return value is undefined.
	int MaxPossibleSSValue()	Returns the maximum possible standard semantics value approximated by this approximating domain.
	int MinPossibleSSValue()	Returns the minimum possible standard semantics value approximated by this approximating domain.

Table 15. Methods in Interface *IGuidingIntDomainInfo*.

The *IGuidingIntDomainInfo* value domain interface serves as the integer/boolean-specific approximating domain interface for values. As indicated in Table 15, operations are divided into two groups: Those applicable to abstract booleans and those relevant for abstract integers.

Abstract booleans are used in predicates for loops and conditionals, and consequently must be capable of indicating to the analysis machinery whether or not they are known to be true or false. (For example, in order to perform the abstract interpretation of a conditional, it is desirable for the analysis precision to be able to recognize cases in which the predicate is known at analysis time to be true or false).

[21] For example, without direct access to the value returned by the approximating domain, the code computing the return value in the collecting domain could simply manually recompute the value returned by the approximating domain by explicitly calling the appropriate method on the approximating domain.

The methods within *IGuidingIntDomainInfo* for bounding the extent of integers are not used as directly by the analysis machinery as those for booleans, but are useful when implementing the abstract state and pointer domains. For example, in the pointer value domain, knowledge of the precise or possible range of referents resulting from applying an offset to a pointer can greatly aid in bounding the possible effects of a pointer write. Bounding these effects is frequently an important prerequisite for precise analysis, and knowledge of the magnitude of the offset is a key component in this regard. In the *AbstractState* domain, bounding of integer values can offer substantial benefits in modeling array allocation. For example, if the integral value specifying the size of an array to be allocated is precisely known, a high-fidelity approximating state domain may choose to simulate the contents of that array in a disaggregated manner. The *SSValue* routine would be called in order to extract the (known) allocation length of the array and to allow for allocation of a corresponding model. Similarly, knowing the maximum and minimum possible values of a value may allow for *partial* disaggregation of array elements.

Method	**Description**
IAbstractIntValue ReadInt()	Returns the abstract integer value at the location or locations possibly pointed to by this pointer domain. Note that if more than one location is possibly pointed to by this domain, the ⊔ of the contents of all such locations is returned.
void WriteInt(IAbstractIntValue v)	Writes the specified abstract integer value to the location or locations possibly pointed to by this pointer domain. If more than one referent is possible, a "weak write" is performed, in which the specified value is ⊔ with the contents of each possible referent location.
IAbstractPtrValue ReadPtr()	Similar to *ReadInt*, but reads a pointer value instead.
void WritePtr(IAbstractPtrValue v)	Similar to *WriteInt*, but writes a pointer value.

Table 16. Methods in Interface *IGuidingPtrDomainInfo*.

5.3.2.3 IGuidingPtrDomainInfo

The pointer-specific domain interface for values (shown in Table 16) requires implementation of only four closely related but unusual methods. In particular, they are distinguished by the fact that although they are defined in a *domain*-level interface, they operate on *values* (i.e. bundles of domain-level information). This apparent impedance mismatch results from a characteristic of the architecture mentioned above. Within TACHYON, it is typical (and sometimes required) to jointly define the state and value domains for a given id. These components work together to implement the rules associated

with particular abstract semantics. As discussed in the next chapter, the quantum of data storage for the approximating state domain is the *value* rather than the domain. It is therefore natural that the pointer domain analogue to the approximating state – namely, the approximating pointer domain – be required to manipulate entire values as well.

The memory access routines shown in Table 16 are not called *directly* by the framework, but are instead called from within the abstract state methods responsible for performing pointer reads and writes.

```
public _IAbstractPtrValueReadPtr(_IAbstractPtrValue loc, int srcLoc)
{
    // ok, first we get the value from memory
    _IAbstractPtrValue vUnfiltered = this.NarrowedCurrentGuidingDomain().ReadPtr(loc.TypeSpecif
icGuidingDomain(), srcLoc);

    // ok, now we filter it
    return ((_IAbstractPtrValue) this.FilterTwoValueReadOrWriteOperation(s_ptrReadLocationTypeS
pecificDomainFilter,  (_IAbstractValue) loc, (_IAbstractValue) vUnfiltered, -1, srcLoc));
}
```

Figure 35: Code to Approximate Pointer Reads within the Abstract State.

```
public _IAbstractPtrValueReadPtr(_IGuidingPtrDomainInfo loc, int srcLoc)
{
    return(loc.ReadPtr());
}
```

Figure 36: Sample Implementation of the Pointer Read Routine in a Approximating State Domain.

5.3.2.4 IGuidingDoubleDomainInfo

The methods included in interface *IGuidingDoubleDomainInfo* are similar to those of the integral approximating domain, but do not include the boolean methods *FKnownTrue* and *FKnownFalse*. Any double approximating value must be capable of bounding its range and specifying its precise value in those cases where it is known. The ability to rely on the presence of these operators allows collected domains to scale to handle cases in which the values being manipulated are not fully known.

Method	Description
double SSValue()	Applies only to concrete values.
double MaxPossibleSSValue()	Must be handled by any type of value – concrete or not.
double MinPossibleSSValue()	Must be handled by any type of value – concrete or not.

Table 17. Methods in interface IGuidingDoubleDomainInfo

5.3.2.5 Summary

This section has examined the interfaces that must be implemented by the value *approximating domain*. Because such domains exercise direct control over the abstract states encountered during analysis and (in particular) over the flow of analysis through the program, they are must support additional methods not

required in the other domains. Of particular note in this respect are methods that provide information on the concrete value or potential range of concrete values that are approximated by the domain element, and methods for writing and reading entire encapsulated values to and from memory locations. In addition, the type-generic approximating interface holds methods allowing other domains access to the most recent value returned from the current domain. Notably absent in the approximating semantic value interfaces are value creation and combination methods – the routines that constitute the bulk of the methods present in the general (i.e. not particular to the approximating semantics) value domain interfaces.

5.4 Value-Level Interfaces

5.4.1 Introduction

The previous two sections have presented the interfaces that must be implemented by the user in order to create custom domains collecting user-specified information. Such information is of obvious direct importance to a user who wishes to create a custom abstract semantics. As was noted in the previous chapter, however, the analysis machinery itself manipulates all program quantities at the level of *abstract values* and *abstract states* rather than at the level of abstract value domains or abstract state domains. And while the implementations of the value-level interfaces eventually desugar all methods into calls to domain-level value methods, describing only the domain-level value interfaces tells only part of the story. This section examines the value-level interfaces, while the next section provides an overview of the connection between the two levels of interfaces.

Before launching into the presentation of the interfaces, it is worth pausing to consider an important aspect of those interfaces. Because abstract values are simply sequences of (tagged) domain elements, most operations on values can simply be directly and uniformly applied to the component domains. Some operations, however, require special care. There are three primary types of exceptions to this rule:

- **Methods designed to create values.** As discussed above, class-level value creation methods (such as *CreateUninitialized*) are applied to singletons during execution to allow for execution with a consistent set of domains[22]. Thus, methods used to create an initial value from component domains are not used *during* analysis, because once a single prototype exists for a state we can operate on the level of values. However, in order to *construct* a prototype singleton, we need to be able to build up the value from a specified set of domains.

[22] Despite this machinery, TACHYON currently supports dynamic extension or modifications of the domain sequence during analysis. This is probably an instance of overengineering with no compelling motivation behind it, and should be shelved.

- **Methods applying only to the approximating domain**. For example, in order to determine which branch or branches of a conditional to evaluate, the analysis must determine the approximating domain's level of knowledge concerning the predicate value. To do this, it queries the abstract value that results from evaluating the predicate using the methods *FKnownTrue* and *FKnownFalse*. Such methods are passed on to the approximating domain but not to other domains.

- **Methods used to directly access constituent domains.** The method *GuidingDomain* returns the distinguished approximating domain within an abstract value. This method is used both by the framework itself and by other domains to access the information stored in the approximating domain. For example, a domain whose current value is entirely determined by the current state of the system can consider itself converged in a loop if the contents of the approximating domain have converged as well.

The convention used in illustrating the value interface methods in Table 18, Table 19 and Table 20 is that a blank description indicates that the method's value-level implementation simply calls a corresponding routine in one of the domain-level interfaces.

5.4.2 IAbstractValue

Method	**Description**[23]
IAbstractValue FiniteHeightLUB(IAbstractValue vOther, boolean[] fChangedOut)	
IAbstractValue LUB(IAbstractValue v)	
boolean FIsApproximatedBy(IAbstractValue a)	
boolean FPointer()	Indicates whether this is a pointer abstract value directly.
boolean FIsTopElt()	
IAbstractValue TopElt()	
IAbstractValue BottomElt()	
int CtDomains()	Computes internal domain count.

[23] Blank if analogous method is simply called at domain level.

IDomainInfo GuidingDomain()	Just returns domain directly.
IGuidingValueDomainInfo GuidingDomainGuidingInterface()	Narrowed version of above.
IDomainInfo Domain(int i)	Returns the domain at index *i*. (Note that the index is arbitrary, except for the approximating domain, which is guaranteed to be at index 0).
String Unparse()	
void InstantiateFromDomain(IDomainInfo i)	Allows instantiation of a value from a single domain.
void AddDomain(IDomainInfo i)	Adds another domain to an already existing value.

Table 18. Methods in Interface *IAbstractValue*.

Table 18 shows the methods present in the *IAbstractValue* interface, which serves as the type-generic interface for all values. Most of the methods are simply desugared into corresponding domain-level methods, while a few provide information on internal domains in the aggregate and on a domain-by-domain basis. It should be noted that the *CtDomains* and *Domain* methods are in fact used jointly as an iteration pair – aside from the distinguished approximating domain (which is always located at index 0), there is no way to statically predict at exactly what offset a domain element with a particular domain id will be located.[24] In the current implementation, even a *dynamically* determined index cannot be relied upon to remain valid throughout execution. In the interest of efficiency it is expected that this generality will be curtailed in preference for a system in which the domains are set dynamically but remain fixed throughout analysis.

5.4.3 IAbstractIntValue

	Method	**Description**[25]
Integer	IAbstractIntValue operatorMod(IAbstractIntValue v)	
	IAbstractIntValue operatorAdd(IAbstractIntValue v)	

[24] Note that while the particular sample implementation of the abstract value and abstract state provided in TACHYON and illustrated elsewhere in this thesis happens to keep domains in an order sorted by domain id, this is an implementation detail that should not be relied upon when interacting with abstract values or states.

[25] Blank if analogous method is simply called at domain level.

	IAbstractIntValue operatorSub(IAbstractIntValue v)	
	IAbstractIntValue operatorMul(IAbstractIntValue v)	
	IAbstractIntValue operatorDiv(IAbstractIntValue v)	
	IAbstractIntValue operatorArithAnd(IAbstractIntValue v)	
	IAbstractIntValue operatorArithOr(IAbstractIntValue v)	
	IAbstractIntValue operatorArithXor(IAbstractIntValue v)	
	IAbstractIntValue operatorRightShift(IAbstractIntValue v)	
	IAbstractIntValue operatorLeftShift(IAbstractIntValue v)	
	IAbstractIntValue operatorArithNot()	
	IAbstractIntValue operatorNeg()	
Integer and Boolean Operators	IAbstractIntValue operatorEQ(IAbstractIntValue v)	
	IAbstractIntValue operatorNE(IAbstractIntValue v)	
	IAbstractIntValue operatorLT(IAbstractIntValue v)	
	IAbstractIntValue operatorLE(IAbstractIntValue v)	
	IAbstractIntValue operatorGT(IAbstractIntValue v)	
	IAbstractIntValue operatorGE(IAbstractIntValue v)	
Boolean Operators	IAbstractIntValue operatorLogicalAnd(IAbstractIntValue v)	
	IAbstractIntValue operatorLogicalOr(IAbstractIntValue v)	
	IAbstractIntValue operatorLogicalNot()	
	boolean FKnownTrue()	Simply translates into the appropriate call to the approximating domain.

	boolean FKnownFalse()	Simply translates into the appropriate call to the approximating domain.
LUB Routines	IAbstractIntValue FiniteHeightLUB(IAbstractIntValue v, boolean[] fChangedOut)	
	IAbstractIntValue LUB(IAbstractIntValue v)	
Misc. Methods	String Unparse()	
	boolean FIsApproximatedBy(IAbstractIntValue v)	
	IGuidingIntDomainInfo TypeSpecificGuidingDomain()	Directly returns a narrowed reference to the approximating domain.
Creation	IAbstractIntValue CreateFromConstant(int v, int srcLoc)	
	IAbstractIntValue CreateUninitialized(int srcLoc)	

Table 19. Methods in Interface *IAbstractIntValue*.

Table 19 illustrates the methods contained within the *IAbstractIntValue* interface. There are few surprises in this interface – almost all of these methods are translated into corresponding methods at the domain level. Two exceptions to this are the methods for determining the truth status of a boolean and a method for obtaining the approximating domain element. The boolean methods are used in determining the flow within the analysis of a conditional or loop, and consult only with the approximating value domain. The method for obtaining the approximating domain element can be used by the framework and domain other domain implementations in order to make use of information regarding the set of possible concrete values approximated by the current value.

5.4.4 IAbstractPtrValue

	Method	**Description**[26]
Combination	IAbstractPtrValue operatorOffset(IAbstractIntValue i)	
	IAbstractIntValue operatorSub(IAbstractPtrValue v)	

[26] Blank if analogous method is simply called at domain level.

	IAbstractIntValue operatorEQ(IAbstractPtrValue v)	
	IAbstractIntValue operatorNE(IAbstractPtrValue v)	
	IAbstractIntValue operatorLT(IAbstractPtrValue v)	
	IAbstractIntValue operatorLE(IAbstractPtrValue v)	
	IAbstractIntValue operatorGT(IAbstractPtrValue v)	
	IAbstractIntValue operatorGE(IAbstractPtrValue v)	
LUB Operators	IAbstractPtrValue LUB(IAbstractPtrValue v)	
	IAbstractPtrValue FiniteHeightLUB(IAbstractPtrValue v, boolean[] fChangedOut)	
Misc	IGuidingPtrDomainInfo TypeSpecificGuidingDomain()	Directly returns a narrowed reference to the approximating domain.
	Boolean FIsApproximatedBy(IAbstractPtrValue v)	
Value Creation	IAbstractPtrValue CreateFromAllocation(IMemoryRegion region, int srcLoc)	
	IAbstractPtrValue CreateFromAddressOfVariable(IMemoryRegion region, int iOffset, int srcLoc)	
	IAbstractPtrValue CreateUninitialized(int srcLoc)	

Table 20. Methods in Interface *IAbstractPtrValue*.

The methods of *IAbstractPtrValue* are shown in Table 20. With the exception of a narrowed version of the operator returning a reference to the approximating domain contained within the value, all methods are simply applied to the included domains. Note in particular the absence of methods relating to memory access; as show in Figure 35, the memory access code associated with the abstract state

implementation calls directly to routines in the *IGuidingPtrDomainInfo* interface. It could reasonably be argued that this is aesthetically disagreeable and that for the sake of maintaining consistent layering intermediate memory access operators should be placed in *IAbstractPtrValue* which simply delegate their handling to the approximating domain.

5.4.5 *IAbstractDoubleValue*

	Method	**Description**
Combination Operators	IAbstractDoubleValue operatorAdd(IAbstractDoubleValue v)	
	IAbstractDoubleValue operatorSub(IAbstractDoubleValue v)	
	IAbstractDoubleValue operatorMul(IAbstractDoubleValue v)	
	IAbstractDoubleValue operatorExp(IAbstractDoubleValue v)	
	IAbstractDoubleValue operatorDiv(IAbstractDoubleValue v)	
	IAbstractDoubleValue operatorNeg()	
	IAbstractDoubleValue operatorMap(IDoubleMap fn)	
Comparison Operators	IAbstractIntValue operatorEQ(IAbstractDoubleValue v)	
	IAbstractIntValue operatorNE(IAbstractDoubleValue v)	
	IAbstractIntValue operatorLT(IAbstractDoubleValue v)	
	IAbstractIntValue operatorLE(IAbstractDoubleValue v)	
	IAbstractIntValue operatorGT(IAbstractDoubleValue v)	
	IAbstractIntValue operatorGE(IAbstractDoubleValue v)	
Misc. Operators	boolean FIsApproximatedBy(IAbstractDoubleValue v)	
	IAbstractDoubleValue FiniteHeightLUB(IAbstractDoubleValue v, boolean[] fChangedOut)	
	IAbstractIntValue operatorConvertToInt()	
	IAbstractDoubleValue LUB(IAbstractDoubleValue v)	
	IAbstractDoubleValue GLBWithTruePredicate(IAbstractIntValue vPredicate)	
	String Unparse()	
	IGuidingDoubleDomainInfo TypeSpecificGuidingDomain()	
Creation Operators	IAbstractDoubleValue CreateFromConstant(double v, int srcLoc)	
	IAbstractDoubleValue CreateUninitialized(int srcLoc)	

Table 21: Methods in Interface *IAbstractDoubleValue.*

The methods associated with the value-level double precision interface are shown in *IAbstractDoubleValue*. With the sole exception of the method *TypeSpecificGuidingDomain*, all operations performed at the value level are transparently desugared into calls at the domain level. The set of operations contained in this interface is thus essentially the same as that associated with the domain level interfaces for doubles.

5.5 Taming Lower-Level Languages

Lower-level languages such as ANSI C or assembly language contain many features that are difficult to model compactly and consistently within a "compiled analysis" framework. The chief problem here is not so much the weak typing or low-level character of built in types (although these characteristics can substantially add to the overhead of precise analysis), but the fact that such languages frequently make little or no assurances of data atomicity. As a result, there are multiple ways of accessing the *same* information at different levels of granularity (for example, writing one data type, and then reading from the internals of that data type at a later point), and "clumped together" in different ways (for example, writing a series of bytes and then later reading them out as a single unit). Three examples of such difficulties are given below:

- **Union constructs.** C language union constructs have traditionally been used as a means of interchanging data stored in different types. Thus, a program might maintain a union containing both character arrays and structures. When a file is read from a persistent store, the byte stream could be placed into the character arrays, and then read out as structures.

- **Lack of Heap Type-Safety**. In C, the heap is allocated by means of type-generic function calls and is not fixed to capture information regarding only about a particular type. Not only can heap accesses be performed without type checks, but they can also be made in violation of the atomicity of stored data types.

- **Weakly Typed Pointers.** Weakly typed pointers are one of the most difficult aspects of low-level languages to model. Type casting from one pointer type to another can be used to allow pointers to "reach inside" certain data types and extract pieces of those data types for processing. By means of pointer arithmetic, such accesses can be performed at any point within or between written quantities. Finally, pointers can open a Pandora's box of modeling difficulties by virtue of their capacity to take advantage of imprecisely defined aspects of program behavior or the language specification. For example, within certain C language programs it is not unusual for pointers to be used to probe into the stack in unconventional ways (e.g. to gain customized access to variable numbers of arguments on the stack).

The lack of "access atomicity" in a language greatly complicates the modeling of data within that language. While an ideal compilation scheme would allow for a direct modeling of standard semantics data objects by corresponding objects in the abstract semantics, this does not seem feasible within a language lacking access atomicity. (For example, while each constituent byte within a floating point

number can be read using a union or pointers, it does not seem feasible to build up every abstract floating point value directly out of abstract bytes.)

The alternative is to maintain sufficient bookkeeping mechanisms and superstructure to allow "interception" and translations of all data accesses into corresponding accesses within the abstract semantics. For example, a write to even a precisely known single byte of a larger quantity would be intercepted and modeled as invalidating that quantity, and the read from any subcomponents of a larger quantity would be recognized as such and would return the top element of the appropriate type domain.

While such "canonicalization" mechanisms are required for accurate and sound modeling of the standard language semantics within the abstract semantics, they are accompanied by increased analysis complexity and resource demands. It remains to be seen whether that advantages of a truly high-precision analysis scheme remain compelling in the context of such additional expenses.

5.6 Conclusion

This chapter has examined the interfaces associated with abstract values, and some of the issues involved in implementing the mapping from the value-level interface to domain-level methods. For the most part, the make-up of the interfaces is straightforward and unsurprising, although a few methods have warranted additional comment and discussion. As is the case with the abstract state, the current domain implementation offers substantial flexibility to the user in implementation and during analysis, by requiring the implementation of fewer interfaces by the user and by permitting the addition of domains while analyses are operating. Unfortunately, this generality imposes substantial performance costs for each value combination. Fortunately, adoption of a more restrictive but efficient system would only require very few, extremely localized changes.

Chapter 6 Interfaces for the Generic State Domains

6.1 Introduction

The previous chapter presented the interfaces associated with value domains and the analysis interface. This chapter performs a similar function for the state domain. Although the state interface lacks the typing issues that complicate the value interface, it is itself made more involved by the need to distinguish between a distinguished "current state" and all other states.

6.2 Two Approaches to Modeling the State

6.2.1 Abstract State as Object

Reifying analysis state as an object creates an abstraction of program state that centralizes all accesses to program state and admits to any of a wide range of possible implementations. This encapsulation of the abstract state as an object gives rise to a diverse set of practical benefits. Three of the most important of these benefits are mentioned below.

- **Centralized Management.** Unlike concrete execution, abstract execution must deal with uncertainty in the path taken by the program at runtime. As will be discussed further in Chapter 7, this requires the analysis to frequently shift the point of current execution between different abstract states. This requirement in turn necessitates that the abstract state mechanisms be capable of "saving away", taking the least upper bound of two states, and instantiating previously stored states. Consolidating all state mechanisms in a single object would simplify this process.

- **Centralized Instrumentation.** The state implementation serves as a natural point for adding user-customized instrumentation for measuring state activity, and a natural point of storage for the statistics collected by such instrumentation. For example, such a mapping would provide a very simple manner of implementing traps on accesses to particular memory locations or ranges of locations.

- **Computational Cost.** By allowing a variety of implementations of the primitives for the access and allocation of locations within the state, the state abstraction permits great flexibility encoding the values stored within the state. For example, even while operating in the standard semantics, a state object could store values in a compressed format – for example, by run-length encoding an array with uniform elements. Alternatively, the state could treat all locations outside of a relatively small cache

as containing [NO85] $\top$ without the need to explicitly simulate the space that they take up. As a third alternative, it would be easy for a state implementation to shift between a regime in which the values in program locations are monotonic in the value lattice (such as that seen in [Chen 1994]) and one in which program locations are allowed to vary more generally (such as that in [Osgood 1993]).

6.2.2 Advantages of Decentralization

Modeling state as an object and all state references or memory management requests as method invocations allows for easy, unified instrumentation of all state-related accesses. While modeling the program state as a single object has a certain pristine logic to it, it represents a significant departure from the actual manner in which program state is actually used. In particular, from the point of view of software, program state is accessed as a collection of disjoint values rather than through a single object. As an alternative to the "state object" approach, it seems natural to consider a strategy that would preserve the typical pattern of usage by mapping program quantities in the standard semantics code to comparable program quantities within the compiled analysis code. For the purposes of this section, we will term this approach the "direct mapping" strategy. For example, this approach would model formal and local variables in the program being analyzed with formal and local variables in the analysis program, arrays in the source code with arrays in the analyzing program, etc.

A direct mapping approach has two primary benefits: Greater transparency and efficiency. Each of these motivations will be briefly discussed below, with the emphasis on efficiency.

6.2.2.1 Transparency

By its nature, the direct mapping approach creates an analysis executable more closely resembling the user's source code. For example, rather than translating all variable activity into opaque calls to the state object, the variable manipulation is roughly recognizable in the analysis code. This makes the "direct mapping" approach easier to understand and debug.

6.2.2.2 Efficiency

Mapping all state activity into method calls on a general-purpose object necessitates that the activity is approximated using rather general mechanisms within the state. By contrast, keeping a more direct mapping between concrete and abstract state objects allows the compiler which optimizes the analysis code to make use of more specialized resources. A few of the more important of these fortuitous alignments between run-time and compiled-analysis structures are listed below.

- **Run-time Stack and Simulated Stack.** The most important of the state components mappings between the standard and abstract semantics is that between the standard and abstract stacks (that is, between the stacks associated with the original and the corresponding analysis programs.) The ability to map the abstract semantics stack directly onto that used within the standard semantics permits local and parameter variables within the program being analyzed to have as their compiled analogues local and parameter variables in the analyzing program. Similarly, return values within the input program correspond to return values in the abstract program; frame pointers in the original program are realized as frame pointers in the analysis. In addition to substantially reducing the costs associated with activation-record manipulation, the straightforward character of this mapping has the immediate consequence of permitting a very wide range of other program-stack-related structures to operate efficiently. These advantages do not come without a cost, however: the straightforward correspondence between the analysis and machine stacks can somewhat complicate the handling of certain portions of the analysis (e.g. the fixed-pointing of recursive function calls and the modeling of unknown-sized arrays).

- **Global Program Data Dual to Global Data in Analyzed Program**. Just as the standard semantic stack finds its analogue in the stack of the non-standard semantics, global data within the standard semantics maps directly to global data within the non-standard semantics. This permits global data within the program being analyzed to be accessed and manipulated during the analysis without the need for any intermediate structures. This alone can have beneficial performance advantages (e.g. when operating on large global arrays), but has even more substantial consequences when coupled with the ability to map variables in the input program to variables in the program performing the analysis (as is discussed in the next item).

- **Optimization of Resources.** In addition to stripping away layers of indirection, a more direct modeling strategy allows the application of standard compiler optimizations to the resource allocations discussed above. For example, the compiler may choose to register allocate the references to commonly used variables in the analysis, just as it would the variables being modeled in the original source code

The direct and indirect effects of eliminating layers of overhead are attractive from an efficiency standpoint. Unfortunately, these advantages come only at the cost of complicating semantic parameterization and user-defined state management. The next section discusses a compromise strategy.

6.2.3 *State Modeling: A Hybrid Strategy*

The section above provided a brief overview of two different approaches to state modeling. The first of these approaches encapsulated the state as a single, centralized object. By contrast, the "direct mapping" strategy modeled individual state objects in the standard semantics by corresponding objects in the abstract semantics. The state object had the strong advantages of easy semantic parameterization and instrumentation, while the direct mapping strategy had the virtues of greater transparency and efficiency. This section surveys a compromise approach adopted by TACHYON that tries to combine the best of each of these strategies.

In particular, TACHYON divides an abstract state into two pieces: A single component implementing the *AbstractState* interfaces, and a set of local, formal, and global variables of type *AbstractValue* that serve as approximations to most program variables. "Escaped variables" (variables to which an "address of" operator is applied in the program text) constitute an important exception to this separate handling. In order to permit access to escaped variables through pointer mechanisms, such variables are instead modeled using the *AbstractState* object rather than as distinct variables in the translated program. Examples of this use of both escaped and non-escaped variables are given in the sections below. Although seemingly awkward, this division of an abstract state into two pieces has several advantages:

- **Semantic Parameterizability and Ease of Instrumentation.** The analysis system maintains the semantic parameterizability of the abstract state by performing state operations at two levels: One method call for the abstract state object itself, and a set of method calls or actions[27] for the appropriate variables. For example, the least upper bound operator associated with a control flow join would be applied both to the abstract state and to each variable in turn. In cases where instrumentation seems particularly likely (e.g. for memory accesses), the state object is also allowed to "filter" the variable accesses. (See, for example, Table 23 and subsequent discussion.) By virtue of this careful mirroring, the state object can serve as the centralized repository of information and instrumentation.

- **Transparency.** The use of abstract variables as approximations to variables in the standard semantics is both intuitive and visually obvious.

[27] Not all such actions entail dual method calls: For example, storing away a state being temporarily deactivated necessitates both a method call to the abstract state object and a simple copying of the reference to the associated variable: See Chapter 8.

- **Efficiency through Elimination of Layering Overhead.** By representing variables independently, the system can bypass multiple layers of overhead associated with memory accesses and manipulate those variables directly.

- **Efficiency through Specialization.** Variables differ from allocated memory most notably in that their quantity, size, and ordering are statically known. This allows for the elimination of the interpretive overhead implicit in the more general memory access mechanisms through the use of specialized variables whose handling is hardwired to use the static information.

- **Reflection of Different Convergence Concerns.** Because the set of variables for a program is of fixed size, the variables are associated with a joint abstract domain of bounded height and cannot prevent analysis termination. In contrast to typical memory modeling, there is no need to maintain bookkeeping machinery to ensure convergence of the variable domain. Separate handling of variables and heap structures reflects this difference and allows for cheaper variable modeling.

When reviewing the interfaces below, the reader should bear in mind the fact that the system does use a hybrid strategy, and that non-escaped variables will not be directly affected by the methods given. Further details on the handling of such variables are deferred until Chapter 8, which surveys how the analysis mechanisms work in tandem to apply state operations both to the state object and to the non-escaped variables.

6.3 State Taxonomy

The conceptual model underlying the interfaces presented in this section classifies states into two categories. This state ontology is reified in the interfaces. This section first briefly explains the distinction that is drawn and its conceptual underpinnings. The section then continues to on to discuss some of the advantages and disadvantages to this decision as it affects potential strategies for implementing the state domain.

6.3.1.1 The Abstract Execution Graph

When run under the standard semantics, the program evolves according to precisely defined execution sequences. By contrast, in an abstract semantics the path of execution at run-time is not always clear: For example, it may not be known which branch of a conditional execution will flow at runtime. Thus, while a particular state in the standard semantics will always have a unique "next state"; this is not true in the abstract state.

Within imperative programs, the notion of program execution is closely bound up with that of state: Program evolution occurs by mapping one state to another. At any given moment in time, there is a single distinguished "current state" that marks the current point of execution of the program. By contrast, because the path taken by an analysis through the program will usually be non-linear due to control-flow splits, the notion of the "current state" concept loses its precise meaning in the abstract semantics. In the abstract semantics, the loci of execution are divided between many distinct states (e.g. those associated with each branch of the conditional).

6.3.1.2 Linearization and Symmetry Breaking

Because its analysis runs in a single-threaded manner, however, the TACHYON performs abstract execution on a *linearized* form of the non-linear execution graph. As illustrated by the numeric labeling in Figure 37, this linearization imposes a partly arbitrary ordering on all abstract state operations. The resulting analysis will typically not be monotonic with respect to the time scale of the standard semantics. Handling this non-monotonicity requires the analysis machinery to "back up" to previously encountered states and to represent multiple states at a single point. (For example, as shown in Figure 37, the analysis must maintain the starting state for the "false" branch of a conditional while evaluating the "true" branch.) The need to represent multiple states imposes substantial costs on abstract execution that are not shared by its standard semantics counterpart.

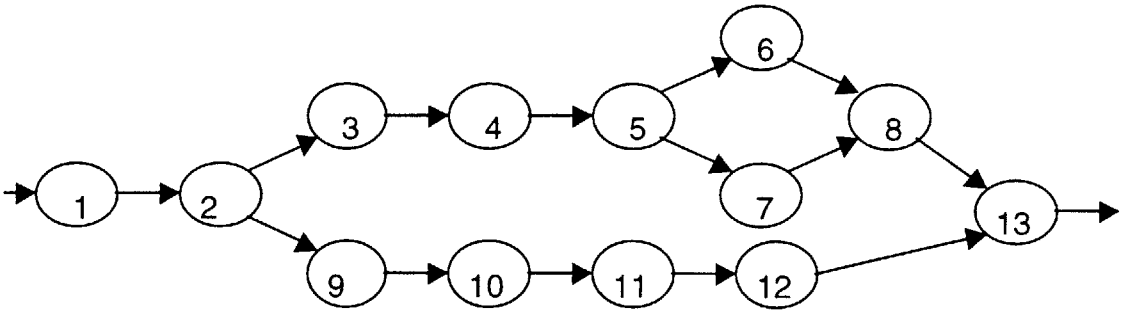

Figure 37: Linearization of a State Flow Graph for the Abstract Semantics.

Although control flow splits render the analysis non-monotonic and blur the notion of the "current" state from the point of view of the *standard* semantics, the abstract execution is associated with its own time scale (a characteristic suggested by the labeling of Figure 37). We can term the unique abstract state in effect at each point along this analysis time scale the "current abstract state" for that point.

6.3.1.3 Linearization: Advantages and Disadvantages

This linearization has both positive and negative potential performance consequences. On the positive side, maintaining only a single dynamic state allows straightforward cost savings by representing the current state in a specialized, disaggregated manner, and expressing all other states relative to the current state. (This technique is discussed further in Chapter 12). This fully instantiation of the abstract current state allows for highly efficient handling of operations applied to the state and mirrors the fully instantiated character of the current state in the standard semantics.

The downside of linearization is that it eliminates the possibility of extending the system to evaluate coexisting lines of execution in parallel. While all analysis algorithms known to the author operate in a single-threaded, linearized manner, a less restrictive – and perhaps wiser –choice of interfaces would have avoided linearization in favor of allowing for multiple contemporaneous current states. This would leave open the option of parallelizing the analysis while still allowing for the "current state" technique discussed in Chapter 12 to be used for implementation until the point where a new, parallel implementation architecture is adopted.

6.3.1.4 Conclusion

This section has examined the background for TACHYON's practice of identifying a unique, disaggregated "current abstract state" at each point during abstract execution. (Henceforth simply termed the "current state" in contexts discussing abstract execution.)

This practice has its origins in the linearization of a naturally non-linear execution graph for the sake of abstract analysis. The concept of the current state plays an important role in the interfaces defined below. Reification of the current state/non-current state distinction in the interface allows state domains to represent that state in a highly efficient manner, but has the disadvantage of ruling out future parallelization of the analysis.

6.4 Modeling Control Flow

The value domain interfaces discussed in the last chapter had methods associated with each variety of construct (expressions) that manipulates values. A straightforward approach to modeling control flow

would employ a similar strategy for states, in which the state domain interface would contain methods for each type of construct (statements) that operates on states. The naïve approach here is unworkable: Unlike (most[28]) expressions, statements contain subcomponents that may or may not be evaluated, and it is not possible to simply duplicate the style of the expression interfaces for states.[29]

Rather than associating methods for each type of statement, the TACHYON system instead models all statements as combinations of a small basis set of primitive control flow operations. In particular, the control flow associated with statements is expressed as a combination of the following elements:

- **Suspension of the current state.** The current state is retired from serving as the current state, and reorganizes itself to be saved away for a later reactivation or join with a current state.

- **Continuation of a previously suspended state.** Here, a state that was previously suspended and saved away is reintroduced as the current state, and continues execution.

- **Dynamic Split.** The current state is split into two "branch" states based on the value of a predicate. One the branches continues if the predicate holds a certain value.[30] Otherwise, the other branch continues.

- **Join of a previously suspended state with the current state.** In this case, a previously suspended and saved-away state is combined with the current state. The result then continues execution. This is most frequently used for the join at the end of a conditional.

- **Join of the current state to a previously suspended state, and continuation from that state.** The current state joins a previously saved-away state, creates a suspended state of the value at the join, and continues execution. This is most frequently used for continuations from within a loop.

- **Straight-Line Execution.** The current state simply performs the current side effect and continues as the current state.

---

[28] Some programming languages do offer expressions of which only pieces are evaluated: For example, consider the ternary operators "?:" in C and "IIf" in Visual Basic. Were the language being modeled to include such constructs, compilation of expressions into their abstract semantics analogues would have to contain similar mechanisms to those seen in statement handling.

[29] While it might be possible to construct a more sophisticated style of interface in which per-statement methods take abstract state arguments more reminiscent of continuations, this approach has not been investigated.

[30] TACHYON only currently supports conditionals based on boolean predicates, although the modeling of more general forms of dispatch would require little additional work.

These elements are graphically depicted in Figure 38, in which the arrows represent state transitions, circles stand for the current state, and the cylinders indicate storage used to save away states. Each of these control flow primitives is associated with corresponding methods in the state interfaces. We turn now to a discussion of those interfaces.

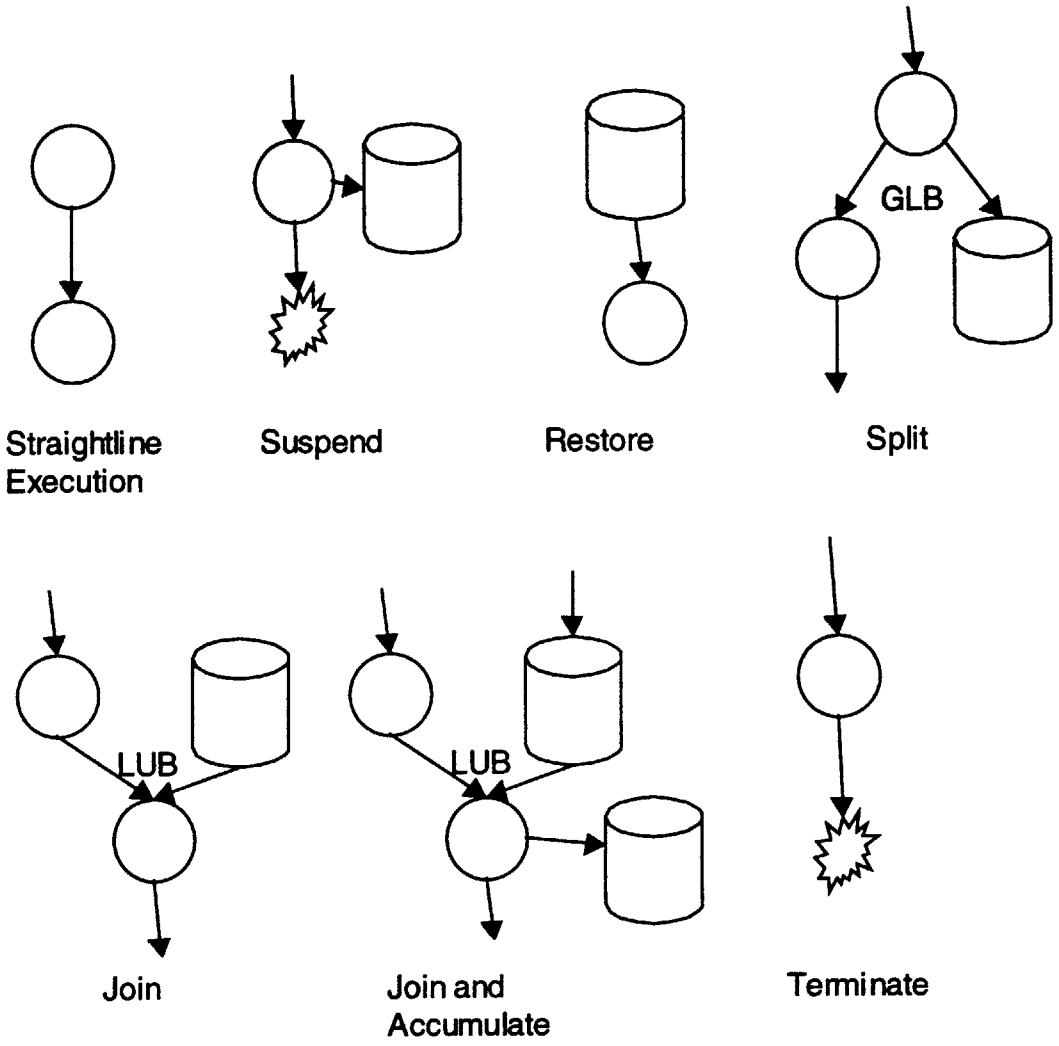

Figure 38: Abstract Control Flow Primitives.

6.5 Domain Interfaces

This section provides an overview of each of the interfaces that must be implemented by state domains. As might be expected from the last chapter, approximating and collecting state domains are associated with somewhat different interfaces. In addition, the domains of the *current* abstract state (whether

approximating or collecting) are required to implement interfaces specific to their position as the current state; these interfaces encapsulate some of the most important functionality associated with states.

6.5.1 IStateDomainInfo

	Method	**Description**
Current	IStateDomainInfo SetAsCurrentState(int srcLoc)	Sets this state to be the distinguished current state. Note that this routine is responsible for Retiring the existing current state by calling that state's *RetireFromCurrentState* method.
LUB	IStateDomainInfo FiniteHeightLUBFromCurrentStateAndSetAsCurrentState(IStateDomainInfo stateCurrAndLHS, int srcLoc, boolean[] fChanged)	Takes the least upper bound of the current state with "this", and returns the result after setting it as the current state. Note that the parameter fChangedOut implements what is in effect a pass-by-reference argument that is used to indicate whether the result of the LUB is higher in the lattice than "this". Note that this routine must guarantee convergence to a fixed point after a finite number of applications. The srcLoc parameter provides information on the location in the code at which the operation is being applied.
	void LUBToCurrentState(IStateDomainInfo domainStateCurrAndLHS, int srcLoc)	LUBs the specified (saved-away) state to the current state and continues to use the result of that LUB as the current state.
Utility	IStateDomainInfo Clone()	
	int DomainId()	Returns the domain id associated with this state. This domain id should be shared with one domain of the integral and pointer types as well in order to allow for initialization of allocated regions.
	boolean FIsApproximatedBy(IStateDomainInfo i)	
	ICurrentStateDomainInfo CurrentInterface()	
	IStateDomainInfo TopElt()	Returns the distinguished top element for this state domain.
	boolean FIsTopElt()	Indicates whether this element is the top domain element.

	String Unparse()	Renders a textual representation of this domain information (for debugging and convenient use by other domains).
Value	IIntDomainInfo CreateCorrespondingIntValueDomainTopElt()	
	IPtrDomainInfo CreateCorrespondingPtrValueDomainTopElt()	
Context	IAbstractState Context()	Retrieve the enclosing abstract state.
	void SetContext(IAbstractState v)	Set the enclosing abstract state.
State	IStateDomainInfo CreateInitialState()	Creates the element of this domain representing the initial value.

Table 22. Methods in Interface *IStateDomainInfo*.

The *IStateDomainInfo* interface must be implemented by any state domain – whether approximating or collecting domain, in the current state or not. Table 22 shows the methods that compose that interface. Particularly notable are the routines used to create the initial state (conceptually a class-level routine, although applied to a prototype singleton of this domain), to take the least upper bound of the current state with the state domain, and to set this domain as the current state. The last routine (encapsulated in *SetAsCurrentState*) is of particular interest in light of the discussion in Section 6.3.1.3. State domain implementations can take advantage of the transition to current state to reorganize the data structures in the state domain in order to optimize the sequential access and incremental modification associated with the current state. The analogue of this routine for *variables* is to simply instantiate the distinguished "current" copy of a variable with the reference to the appropriate saved-away copy.

6.5.2 *ICurrentStateDomainInfo*

	Method	Description
Control Flow	IStateDomainInfo DynamicSplit(IIntDomainInfo vCalculated, int valueConstantToContinue)	Splits the current domain based on the value of a run-time predicate. If vCalculated == valueConstantToContinue at run-time, the current state continues. Otherwise, the state returned by the function becomes the currrent state. Note that because the abstract state does not directly manage the contents of the automatic variables, typically this state method will only be called it is known that the value of the predicate is unknown.
	IStateDomainInfo SuspendState()	Suspends the current state and returns a snapshot of that state that at some point in the future can be LUBbed to the current state or set as the current state.
	void TerminateExecution(int srcLoc)	Represents a definitive termination of the current state (e.g. if this branch of the program halts due to an error or a halt state.)
	Void EnterSourceBasicBlock(int iBasicBlock)	Informs the state domain that execution has reached a certain basic block in the source code. Note that transitions between basic blocks during analysis do not necessarily imply uncertainty regarding the path of execution – Because of information available to the approximating domain, the path of abstract execution may in fact be precisely known. Identification of basic blocks from the source code is merely used as means of unambiguously establishing the point of execution.
	void RetireFromCurrentState(int srcLoc)	Signals that this state will no longer serve as the current state. The existing current state may choose to reformulate its data structures in order to reflect the fact that it will no longer be accessed frequently.
Automatic	Void CreateStackFrame(int cbSizeInBytes, int idFunction, boolean fSingleConcreteStackFrame)	Reflects the creation of a stack frame associated with function *idFunction*, and of *cbSizeInBytes* in length.
	void PopStackFrame()	Pops the most recently created stack frame.
	void CreateVariableLValue(int idVariable, int cbSize)	Notifies the state of an automatic lvalue being created that is associated with the variable of id *idVariable* and is of size *cbSize* bytes.

	void DestroyVariableLValue(int idVariable)	Notifies the state that the lvalue associated with variable of id *idVariable* has been eliminated.
	void PushBlock(int cbSize)	Notifies the state of the pushing onto the stack of an unstructured block of *cbSize* bytes.
	void PopBlock(int cbSize)	Notifies the state of the popping from the stack of an unstructured block of *cbSize* bytes.
Utility	boolean FIsApproximatedBy(IStateDomainInfo s)	
Memory Read/Write Filters	IIntDomainInfo ReadIntFilter(IPtrDomainInfo loc, IIntDomainInfo valueAlreadyRead, int srcLoc)	Allows this state domain to filter an integer value just read from the memory model associated with the approximating state domain. Note that while the entire value (i.e. all domains) have been retrieved from the approximating state domain memory model, this filter is applied only to domain elements drawn from the same domain as this state domain (i.e. the filter is applied only to integer domains sharing the same Domain Id as this state domain). Note also that the pointer domain element used for the read is provided, as is the source code location for the point at which the value is being read. NB: It may well be the case that the value that has just been read is drawn not from a single, precise, location, but represents the least upper bound of reads from many possible locations. (e.g. in cases where a read takes place from a pointer whose referent is imprecisely known.)
	IIntDomainInfo WriteIntFilter(IPtrDomainInfo loc, IIntDomainInfo valueToBeWritten, int srcLoc)	Conceptually similar to the above, but applied to memory *writes* of integral values, and is applied *before* the entire value is written to the approximating state domain memory model. As above, the filter is only applied to domain elements from the same domain as the current state domain. As above, the value to be written may in fact participate in many different locations: If the write is occurring to an imprecisely known pointer, it will be a "weak write" in which case the (already filtered) value to be written will be LUBbed to the contents of each possible referent.
	IDoubleDomainInfo ReadDoubleFilter(IPtrDomainInfo loc, IDoubleDomainInfo valueAlreadyRead, int srcLoc)	Similar to the *ReadIntFilter* except that this is used to filter reads of double values.

	IDoubleDomainInfo WriteDoubleFilter(IPtrDomainInfo loc, IDoubleDomainInfo valueToBeWritten, int srcLoc)	Similar to *WriteIntFilter*, but used to filter writes of double values.
	IPtrDomainInfo ReadPtrFilter(IPtrDomainInfo loc, IPtrDomainInfo valueAlreadyRead, int srcLoc)	Similar to the *ReadIntFilter* except that this is used to filter reads of pointer values.
	IPtrDomainInfo WritePtrFilter(IPtrDomainInfo loc, IPtrDomainInfo valueToBeWritten, int srcLoc)	Similar to *WriteIntFilter*, but used to filter writes of pointer values.
Variable Read/Write Filters	IIntDomainInfo operatorAssignmentToIntVariableFilter (int idVariable, IIntDomainInfo vCurrent, IIntDomainInfo valueToBeWritten, int srcLoc)	Used to filter the value to be written to a *variable*. Note that instead of being provided with a pointer domain associated with the location to be written (as is used above), the routine is given the id of the variable being written to. Note as well that the *current* value of the variable is additionally specified. As for memory read/writes, the location of the operation is also specified.
	IDoubleDomainInfo operatorAssignmentToDoubleVariable Filter(int idVariable, IDoubleDomainInfo vCurrent, IDoubleDomainInfo valueToBeWritten, int srcLoc)	Similar to the above, but writes to a double variable rather than an integer variable.
	IPtrDomainInfo operatorAssignmentToPtrVariableFilter (int idVariable, IPtrDomainInfo vCurrent, IPtrDomainInfo valueToBeWritten, int srcLoc)	Similar to the above, but writes to a pointer variable rather than an integer variable.
	IIntDomainInfo operatorReadFromIntVariableFilter(int idVariable, IIntDomainInfo vAlreadyRead, int srcLoc)	Allows for domain-specific filtering of integer values *read* from a particular value. In addition to the read variable, the identity of the value to be read and the associated source location are also specified.
	IDoubleDomainInfo operatorReadFromDoubleVariableFilter (int idVariable, IDoubleDomainInfo vAlreadyRead, int srcLoc)	As above, but filters *integer* values.

	IPtrDomainInfo operatorReadFromPtrVariableFilter(int idVariable, IPtrDomainInfo vAlreadyRead, int srcLoc)	As above, but filters *pointer* values.

Table 23. Methods in Interface *ICurrentStateDomainInfo*.

The *ICurrentStateDomainInfo* interface (shown in Table 23) is the interface exposed by all domains in *current* states. This interface is the central state interface used during program execution, and many of the methods undergo frequent use throughout the running of the program.

The control flow methods are used to inform the state domains of how the run-time flow of execution relates to the analysis process. Handlers for these methods perform state reorganization and bookkeeping manipulation (for approximating domains), and collect information on program flow (for collecting domains). Note that these include only the control flow methods applicable to the current state; other control flow methods are found in the *IStateDomainInfo* interface discussed earlier.

The notification *EnterSourceBasicBlock* is worth singling out for mention. This method is designed to allow operations invoked in user domains to unambiguously establish the point in the user's program at which they are operating. For example, in order to perform classical constant propagation and folding, we would like to be able to accumulate information regarding the values taken on by each expression within the program. This can be accomplished by having expressions accumulate of code up to the level of at the state interfaces, a technique used by the code generation domain of Chapter 9. A simpler approach is to have each value operator take the least upper bound of its result with previous values of that operator for the same expression. In order to do so, the operator requires unambiguous information as to the expression in the source code for which it is executing. Because all expressions have a precise offset relative to the enclosing basic block, this context can be exactly determined through subscription to the notification *EnterSourceBasicBlock*.

The automatic storage management interfaces allow an analysis to track and approximate the dynamics of global and stack-based storage. If desired, an abstract state can also use the information provided by such routines to construct and maintain a complete external model of program variables.

The two classes of read/write "filtering" routines – one for variable access, the other for memory access – provide a means by which individual domains can modify the domain-specific information just after it is read or just prior to when it is written. The motivation for these routines arises from rather pragmatic considerations. In order to avoid requiring each state domain to maintain similar memory models, the

analysis machinery and the approximating state domain are given complete responsibility for storing and retrieving *entire values* from variables and memory. Unfortunately, removing the need for additional state domains to participate in the retrieval and storage process also eliminates such domains' ability to perform custom actions in response to such memory accesses. Such custom handling of accesses play important roles in many semantic domains – for example, those that compute reaching definitions, profile counts, and the information necessary for copy propagation or displaying program slices. In order to allow non-approximating state domains to perform access-specific routines without forcing them to shoulder the onus of maintaining a sophisticated memory model, we allow such domains to perform custom "filtering" for each type of memory access. Values to be written are filtered on a domain-by-domain basis immediately prior to being written, while values to be read are filtered immediately following their retrieval from memory.

In general, such filtering would typically be desired by *collecting* domains, because such domains are responsible for accumulating information on program behavior. There is, however, an important exception to this rule: Implementing monotonic access semantics. An important avenue towards boosting the speed of analysis (at the cost of analysis precision) is to make the contents of all abstract memory locations rise monotonically in the lattice. This can be implemented by taking the least upper bound of all values written with the pre-existing (initialized) memory contents. Such semantics can be easily realized by implementing the filter routines such that they return the least upper bound of the existing value with the value to be written. While the non-variable methods rely upon the (approximating) state domain's knowledge of the pre-existing contents of the specified location in order to perform the least upper bound, this is not possible with externally simulated variables. The variable-oriented filters therefore explicitly specify both the pre-existing value and the value to be written.

Note that the filtering is carried out such that each state domain filters the information associated with the corresponding value domain in the value that has been read or will be written. If no state domain exists for a particular encountered value domain, the contents of that value domain are passed through unchanged. If no value domain exists for a particular state domain, the value domain is treated as if it is associated with the distinguished $\top$ value for that domain.[31]

[31] Note that this case is not currently supported, although little additional effort would be required for its implementation.

6.5.3 *IGuidingStateDomainInfo*

	Method	Description
Most Recent Return Values	IIntDomainInfo GuidingStateIntReturnValue()	Returns the most recent integral domain value info returned by this domain. This allows other domains to make use of the information returned by the approximating domain for the same operation.
	IDoubleDomainInfo GuidingStateDoubleReturnValue()	Same as above, but for a double value domain element.
	IPtrDomainInfo GuidingStatePtrReturnValue()	Same as above, but for a pointer value domain element.
	boolean GuidingStateBooleanReturnValue()	Same as above, but for a boolean return value (including – most importantly – the boolean conceptually returned by the *FiniteHeightLUBFromCurrentStateAndSetAsCurrentState* routine which indicates whether convergence has been reached). NB: This method reports *boolean* return values, rather than abstract boolean value domain elements.
	IStateDomainInfo GuidingStateStateReturnValue()	Same as above, but reports the most recent *approximating state* domain return value.

Table 24. Methods in Interface *IGuidingStateDomainInfo*.

The methods of the *IGuidingStateDomainInfo* interface are required for implementation by any state – current or not – and are exclusively devoted to providing information on the last value returned by the interface. As was the case for abstract values, such information can be of use to domains that wish to accord their own return values closely with those provided by the approximating domain. For example, a domain implementing constant propagation would find it convenient to monitor the values returned by the approximating domain, in order to recognize cases where a particular expression consistently returns the same value. Alternatively, a code generation domain could inspect the approximating domain return values in order to discover opportunities a deep expression evaluates to a concrete value. In this case, the code generated for the expression could be elided in favor of a simple load immediate instruction. Such routines are not strictly necessary – similar functionality can be achieved by explicitly simulating the approximating domain within each collecting domain that wishes to make use of its information. Providing these routines as part of the standard infrastructure simplifies the creation of such collecting domains and improves performance while imposing little burden on the implementation of the approximating domain.

6.5.4 *IGuidingCurrStDomainInfo*

	Method	Description
Memory Read/Write	IAbstractIntValue ReadInt(IGuidingPtrDomainInfo loc, int srcLoc)	Reads an entire value from a specified memory location. Note that only the approximating pointer domain information is used to perform this read. NB: The information read from the approximating domain model of memory is subsequently filtered on a domain-by-domain basis by each state domain. (See Table 23). In addition to the location to be read, the source location of the read is also provided.
	Void WriteInt(IGuidingPtrDomainInfo loc, IAbstractIntValue v, int srcLoc)	Writes an entire value to the approximating domain memory model. As above, only the approximating domain of the location is used by this routine.
	IAbstractDoubleValue ReadDouble (IGuidingPtrDomainInfo loc, int srcLoc)	Similar to *ReadInt*, except that reads a *double* value.
	Void WriteDouble (IGuidingPtrDomainInfo loc, IAbstractDoubleValue v, int srcLoc)	Similar to *WriteInt*, except that writes a *double* value.
	IAbstractPtrValue ReadPtr(IGuidingPtrDomainInfo loc, int srcLoc)	Similar to *ReadInt*, except that reads a *pointer* value.
	Void WritePtr(IGuidingPtrDomainInfo loc, IAbstractPtrValue v, int srcLoc)	Similar to *WriteInt*, except that writes a *pointer* value.

Table 25. Methods in Interface *IGuidingCurrStDomainInfo*.

The *IGuidingCurrStDomainInfo* interface shown in Table 25 must be implemented by a very particular domain: The approximating domain of a current state. As was discussed in Chapter 5, the approximating state domain of a current state is responsible for maintaining a memory model and managing the memory reads and writes for *all* value domains. Thus, values to be read and written are specified as aggregate values rather than on a domain-by-domain basis. All accesses require information on the location to be accessed (specified by means of an element of the approximating pointer domain), the source location at which the access occurs, and information as to the type of the type of the value being read or written (information implicit in the identity of the method).

As noted above, while the values to be read or written are accessed in an aggregate manner from the memory, they are filtered on a domain-specific basis. This takes place either immediately prior to writing or just following the point at which they are read.

It is important to note that the interface does *not* contain methods for reading and writing to *variables*. As was mentioned in Section 6.2.3, *variables* are managed by the analysis machinery itself rather than any particular domain. The hooks for "filtering" variable accesses lie in the domain *ICurrentStateDomainInfo* rather than in the approximating domain for the current state.

6.5.5 Conclusion

This section has briefly surveyed the abstract state domain interfaces. The approximating domain in abstract states takes over a broader set of special responsibilities than is the case for values: It is responsible for approximating the memory model for *all* domains, and for handling all memory accesses. The abstract domains interfaces presented above thereby permit lighter-weight implementations of non-approximating state domains while still allowing for flexibility in how such domains handle memory accesses.

Rather than placing all state domain methods into a single "fat" state domain interface, the methods applicable to *current* domains alone have been separated out into distinct interfaces. This makes the interfaces somewhat more ordered, although at the expense of somewhat complicating state domain manipulations and a slight performance degradation associated with the frequent downcasting.

6.6 Abstract State interfaces

The sections above have presented the domain-level state interfaces. These interfaces are implemented by the user and encapsulate the semantic rules associated with user domains. Before concluding the chapter, however, it is important to look at another important set of interfaces, namely the state-level interfaces by which the analysis machinery in controls the execution of the abstract semantics program. By contrast to the domain-level interfaces, the state-level interfaces are implemented not by the user but by the TACHYON run-time system. The state interfaces couple on one side to the (program-specific) analysis machinery, and on the other to the user's custom domains (by way of the abstract state domains interfaces discussed in the previous section).

In contrast to the large number of state *domain* interfaces, there are only two state-level interfaces: One interface applicable to all abstract states, and one applicable only for the current abstract state.

As was the case for the value-level interfaces discussed in the last chapter, most state-level interfaces are implemented as calls to the appropriate domain-level interfaces, but there are some exceptions. The tables

displaying the state-level interfaces below reflect this fact. Methods with blank description fields submit to direct translation into calls to domain-level interfaces while exceptions have notes on their operation in this field. Note as well that because variables are modeled externally rather than in the state object, the methods below do not directly apply to values. Instead, wherever some call to a method is made to approximate standard-semantics state manipulation, the analysis machinery takes responsibility of performing analogous operations on values.

6.6.1 IAbstractState

	Method	**Description**
LUB Routines	IAbstractState FiniteHeightLUBFromCurrentStateAndSetAsCurrentState(IAbstractState stateCurrAndLHS, int srcLoc, boolean[] fChangedOut)	
	void LUBToCurrentState(IAbstractState stateCurrAndLHS, int srcLoc)	
Domain-Level Access Routines	int CtDomains()	Returns a count of the domains. Used primarily for iterating through the domains.
	IStateDomainInfo Domain(int i)	Retrieves the domain at position *i*. Note that aside from the approximating state domain (which is guaranteed to be at index 0), we cannot derive the index of a domain from its domain id.
	void AddDomain(IStateDomainInfo i)	Adds a state domain to the AbstractState.
	IStateDomainInfo GuidingDomain()	Returns the approximating domain associated with this AbstractState.
	IGuidingStateDomainInfo GuidingDomainGuidingInterface()	
	void SetDomain(int i, IStateDomainInfo v)	
Set /R	ICurrentAbstractState CurrentInterface()	Returns the current interface to the AbstractState.
	IAbstractState CurrentState()	Conceptually class-level routine. Returns the AbstractInterface of the current state.
	ICurrentAbstractState CurrentStateCurrentInterface()	Conceptually class-level routine. Returns the current interface of the current state.

	void SetAsCurrentState(int srcLoc)	
	boolean FCurrentState()	True only if entire state is the current state.
Utility Routines	String Unparse()	
	boolean FIsBottomElt()	
	IAbstractState Clone()	
State Creation	IAbstractState BottomElt()	
	IAbstractState CreateInitialState()	
	void InstantiateFromDomain(IStateDomainInfo d)	Instantiates a new AbstractState from a single domain. (Note that AddDomain can be called to add additional domains to the state)

Table 26. Methods in Interface *IAbstractState*.

The methods associated with the general-purpose interface *IAbstractState* are shown above. A substantial fraction of the methods serve to provide access to the individual domains within an abstract state, or are class-level methods designed to set or retrieve the distinguished current state. As was discussed in relation to the domain-level methods, the least upper bound routines are commonly used to handle control flow joins. The utility and state creation methods will also be familiar from the domain-level discussion.

6.6.2 *ICurrentAbstractState*

	Method	**Description**
Control-Flow	IAbstractState SuspendState()	
	IAbstractState DynamicSplit(IAbstractIntValue vCalculated, int valueConstantToContinue)	
	void TerminateExecution(int srcLoc)	
	Void EnterSourceBasicBlock(int iBasicBlock)	
	void RetireFromCurrentState(int srcLoc)	
Auto matic	Void CreateStackFrame(int cbSizeInBytes, int idFunction, boolean fSingleConcreteStackFrame)	

	Void PopStackFrame()	
	void CreateVariableLValue(int idVariable, int cbSize)	
	void DestroyVariableLValue(int idVariable)	
	void PushBlock(int size)	
	void PopBlock(int size)	
Utility	boolean FIsApproximatedBy(IAbstractState stateNonCurrent)	
Memory Read/Write	IAbstractIntValue ReadInt(IAbstractPtrValue loc, int srcLoc)	Applied differently to the approximating domains and non-approximating domains. The approximating domain returns the "raw" integral value (possibly collecting it from many locations in the process), while the other domains are given responsibility over "filtering" the corresponding value domain within the returned value.
	void WriteInt(IAbstractPtrValue loc, IAbstractIntValue v, int srcLoc)	Similar to the above, except that the filtering takes place before the value is written.
	IAbstractDoubleValue ReadDouble(IAbstractPtrValue loc, int srcLoc)	Similar in operation *ReadInt*, except that this method reads *double* domains.
	void WriteDouble(IAbstractPtrValue loc, IAbstractDoubleValue v, int srcLoc)	Similar in operation *WriteInt*, except that this method writes *double* domains.
	IAbstractPtrValue ReadPtr(IAbstractPtrValue loc, int srcLoc)	Similar in operation *ReadInt*, except that this method reads *pointer* domains.
	void WritePtr(IAbstractPtrValue loc, IAbstractPtrValue v, int srcLoc)	Similar in operation *WriteInt*, except that this method writes *pointer* domains.
Domain-Level	int CtDomains()	

	ICurrentStateDomainInfo Domain(int i)	
Allocation	IAbstractPtrValue AllocateIntArray(IAbstractIntValue ctElements, boolean fPerformingFixedPoint, int iSrcLoc)	
	IAbstractPtrValue AllocateDoubleArray(IAbstractIntValue ctElements, boolean fPerformingFixedPoint, int iSrcLoc)	
	IAbstractPtrValue AllocatePtrArray(IAbstractIntValue ctElements, boolean fPerformingFixedPoint, int iSrcLoc)	
Variable Read/Write	IAbstractIntValue operatorAssignmentToIntVariableFilter(int idVariable, IAbstractIntValue vCurrent, IAbstractIntValue argRHS, int srcLoc)	
	IAbstractDoubleValue operatorAssignmentToDoubleVariableFilter(int idVariable, IAbstractDoubleValue vCurrent, IAbstractDoubleValue argRHS, int srcLoc)	
	IAbstractPtrValue operatorAssignmentToPtrVariableFilter(int idVariable, IAbstractPtrValue vCurrent, IAbstractPtrValue argRHS, int srcLoc)	
	IAbstractIntValue operatorReadFromIntVariableFilter(int idVariable, IAbstractIntValue vReadRaw, int srcLoc)	
	IAbstractDoubleValue operatorReadFromDoubleVariableFilter(int idVariable, IAbstractDoubleValue vReadRaw, int srcLoc)	
	IAbstractPtrValue operatorReadFromPtrVariableFilter(int idVariable, IAbstractPtrValue vReadRaw, int srcLoc)	

Table 27. Methods in Interface *ICurrentAbstractState*.

The other state-level interface applies to *current* states only. As can be appreciated from the relative sizes of Table 27 and Table 26, the current state interfaces offers much richer functionality than the generic state interface. As in *IAbstractState*, most of the methods are implemented as delegations to the corresponding domain-level routines. The exceptions to this rule are found in the methods to read and write from memory. As was noted earlier, these routines rely upon the approximating domain to perform the memory access for all value domains. Before the write or after the read, the state-level methods additionally pass control to domain-specific "filtering" routines implemented by each state domain. Figure 39 reviews the code used to

implement one of the memory access routines in the abstract state. Because the approximating state domain is responsible for storing the contents of abstract memory location for *all* values, the method implementation delegates the value access for *all* domains to the (current) approximating state domain. The system then allows other state domains to filter the "raw" value read by the approximating state domain before it is returned.

```
public _IAbstractPtrValueReadPtr(_IAbstractPtrValue loc, int srcLoc)
{
    // ok, first we get the value from memory
    _IAbstractPtrValue vUnfiltered = this.NarrowedCurrentGuidingDomain().ReadPtr(loc.TypeSpecifi
cGuidingDomain(), srcLoc);

    // ok, now we filter it
    return ((_IAbstractPtrValue) this.FilterTwoValueReadOrWriteOperation(s_ptrReadLocationTypeSp
ecificDomainFilter,  (_IAbstractValue) loc, (_IAbstractValue) vUnfiltered, -1, srcLoc));
}
```

Figure 39: Handling of a Memory Access Routine in the Abstract State.

6.7 Conclusion

This chapter has examined the interfaces associated with the abstract state object. The interfaces reify a distinction between the *current* other abstract states, thereby allowing for more efficient approximating state operation, but also thwarting potential speedups by exploiting analysis parallelism. The approximating state domain is distinguished by the fact that it is responsible for modeling the contents of abstract memory locations for *all* abstract domains. The core methods associated with memory access are thus preferentially delegated to the approximating domain; other domains are permitted to filter the results, allowing for very easy instrumentation or enforcement of monotonicity in location contents.

The component implementing such interfaces does not approximate *all* aspects of concrete states. In particular, variables that are never the referents of pointers are not directly modeled by the abstract state machinery. For the sake of efficiency and transparency, such variables are instead modeled by the analysis machinery external to the abstract state.[32] The analysis machinery takes care that all actions related to state approximation are applied both to the abstract state object and to each variable. The next chapter illustrates more closely details the modeling of variables and variable/state interaction.

[32] The efficiency motivation for doing this will become clearer in the next two chapters. The advantages arise largely from the fact that static knowledge of variable structure permits the elimination of several layers of abstraction when accessing such variables.

Chapter 7 The Role of Compilation

7.1 Introduction

The past several chapters have discussed the architecture of the TACHYON analysis system, focusing primarily on the interfaces that must be implemented by user-defined domains. Some attention has also been placed on the interfaces by which abstract values and abstract states connect to the analysis machinery. Except for a brief overview in Chapter 4, the nature of that analysis machinery has remained obscure. While the interfaces do presuppose certain broad analysis characteristics (such as the presence of a "current state"), they could be called from any of a variety of analysis implementations – interpreted or compiled, interprocedural or intraprocedural.[33]

This chapter and the one that follows it discuss the novel analysis mechanisms currently in place within TACHYON, which accelerate generic analysis by specializing the generic analysis algorithm to the program being analyzed. This "compiled analysis" is by no means logically required by the interfaces designs presented above, but does help to ameliorate some the performance shortcomings associated with the use of an extensible analysis and has its own advantages as well.

The next section discusses the foundation for the "compiled analysis" approach, as it relates to the discussion of abstract execution presented in Chapter 2. The chapter then continues on to describe the motivations underlying the use of compilation within TACHYON, some of which are obvious and others of which are subtler. This section includes discussion of a rather detailed model examining the performance tradeoffs between a compiled analysis and interpretive analysis in terms of the aggregate user time that they consume. This chapter concludes with a discussion of the asymptotic running time of the analysis algorithm.

The next chapter examines the nuts and bolts of the compilation strategy, and examines how the compiled code dovetails with the interfaces presented in the previous chapters. This chapter begins with a discussion of the modeling of abstract states and values from the point of view of the analysis machinery (issues already touched upon from the user's perspective in Chapter 5 and Chapter 6.) After a brief discussion of some general issues facing compilation (such as the translation of control flow), the text continues on to the

[33] Strictly speaking, the interfaces as defined would also allow for implementation in a worklist framework such as that formulated in [Chen 1994] and [Ayers 1993]. But the assumption of a current abstract state offers little or no advantages into those frameworks, and many implementations that reified the current/non-current distinction would exhibit poor performance in such a framework.

central focus of the chapter: A discussion of how the TACHYON compiler models program constructs. This discussion spans several sections, and includes many compilation templates directly excerpted from the compiler.

7.2 Compiled Analysis: An Overview

Traditional analysis algorithms are specialized to collect a specific type of information on program behavior, but can be applied to any program. The hard-coding of the analysis rules allows for the application of "shortcuts" that increase the performance of the analysis process but inhibits easy extension of the analysis algorithms. (For example, the rules used to guide node visits in worklist algorithms such as that presented in [Ayers 1993] are carefully based on the semantics of the language being evaluated and the type of information being collected.)

By contrast with traditional analysis systems, TACHYON allows the user of specify the character of information on program behavior to be collected during analysis, and the rules governing its collection. Previous chapters have outlined the means in which TACHYON can be parameterized to collect customized information. Unfortunately, this flexibility comes with a serious performance shortcoming: Interpretive overhead. This overhead is reflected most clearly in the dynamic binding of the method calls, but in a more subtle way in the need to make use of a very general abstract interpreter. The commitment to a general interpretive strategy rules out the use of specialized "worklist" algorithms such as that seen in the domain-specific abstract interpretation work of [Chen 1994] and [Ayers 1993].

Fortunately, there is a way to ameliorate some of this performance loss. Chapter 2 introduced the abstract interpreter $I_C^{\#}(s)$: $StateSequence^{\#} \rightarrow StateSequence^{\#}$[34]. We noted in that chapter that for the sake of simplicity, we were describing an abstract interpreter *specific to a particular program P*. Thus $I_C^{\#}(s)$ is more appropriately written $I_{C,P}^{\#}(s)$.

While the mathematical formulation of $I_{C,P}^{\#}(s)$ simplified the discussion of abstract interpretation, most existing[35] analysis abstract execution *implementations* instead make use of an abstract interpreter modeling $I_C^{\#}(P, s)$: $String \times StateSequence^{\#} \rightarrow StateSequence^{\#}$. This abstract interpreter takes as an argument a program P (a string) and an abstract state sequence, and returns the result of evaluating P on s. An

[34] Note that the *StateSequences*$^{\#}$ in the domain and range of the functionality are simply what we have referred to as "abstract states". Because abstract execution is by nature a *collecting* process, they represent *abstractions* of not just individual standard semantics states (as are approximated in the approximating state domain) but of *sequences* of such states (as are approximated by elements of the collecting domain).

[35] An possible exception to this general rule can be found in the recent work on specialization-based run-time code generation, in which a form of analysis (constant folding and subsequent code generation) takes place in a pre-compiled framework. See [Grant, Mock et al. 1997], [Noel, Hornof et al. 1996], and [Leone and Lee 1996] for examples of this increasingly popular approach.

implementation of $I_C^{\#}(P, s)$ has the virtue of generality, but suffers from interpretive overhead in over the course of a large number of analyses on a particular program P.

Taking a cue from Chapter 2, this thesis takes advantage of the fact that simplification through specialization yields not only at the level of mathematical formulation but at an implementation level as well. In particular, by creating an abstract execution mechanism $I_{C,P}^{\#}(s)$ *specialized* to the particular program P, TACHYON eliminates interpretive overhead and gains the performance advantages of hard-wiring knowledge of P into the analysis machinery. Thus, rather than taking the traditional routine of obtaining performance advantages by specializing the analysis algorithm to collect a particular set of information on program behavior, TACHYON streamlines program operation by specializing the analysis to operate a particular program. While this specialization process requires an additional compilation step within the analysis framework, the hope is that that this step will occur relatively infrequently and be localized to a sufficiently small number of modules that the overall benefit of compilation will considerably outweigh its cost. (This cost/benefit tradeoff is explored in some depth in Section 7.3.3). The output of this compilation step is an efficient "compiled analysis" that eliminates the interpretive overhead from the analysis process while retaining the ability to perform an unbounded variety of extended analyses.

7.3 Compilation/Interpretation Tradeoffs

This section examines some of the tradeoffs associated with the use of either a traditional interpreted analysis or a compiled analysis system such as that presented in this thesis. The discussion above focused on *performance* tradeoffs, and much of this section is devoted to a deeper examination of these issues. There are, however, important qualitative considerations at stake as well in the selection of one approach over another.

Before beginning the discussion, it is worth noting that while this section repeatedly contrasts the "compiled analysis" approach of this thesis with a pure interpretive approach, many (standard semantics) interpreters actually make use of some hybrid mixture of compilation and interpretation to accomplish their task. The author is not aware of any use of such a hybrid approach for *analysis*, and many of the advantages claimed herein for compilation would likely apply to such hybrid systems as well. A more detailed examination of the compiled analysis approach would also offer comparisons of TACHYON's approach with such hybrid strategies.

7.3.1 Advantages of Compilation Frameworks

This section touches on a number of advantages offered by the "compiled analysis" strategy relative to traditional analysis strategies.

- **More Accessible Debugging.** The code produced by compilation to an abstract semantics is completely visible to inspection and relatively straightforward to understand. By contrast, the rules underlying the workings of an interpreter-based abstract semantic analysis are hidden within the internals of the interpreter, and are visible only by observing the actions taken by the interpreter. The accessibility of the code implementing the abstract semantics within a compiled system allows for debugging by standard debugging tools, and permits inspection of the subcomponents of each abstract "step". By contrast, an interpreter must provide its own debugging interface. Because an abstract step represents the natural quantum of execution for an abstract interpreter, such an interpreter cannot allow user inspection below the associated level of abstraction.

- **Compact, Componentized Analysis Packaging.** A "compiled analysis" produced by TACHYON is a standalone executable that can perform an unbounded number of analyses on the original program (including analyses that are simply executions of that program on particular sets of inputs). In addition to being attractive from an aesthetic standpoint, this "componentized" packaging is more efficient space-wise than maintaining distinct executables for each type of analysis.

- **Greater Modularity.** Compilation of a program traditionally takes place on a per-module basis. By contrast, interpreters frequently require the complete reloading of a program's source when beginning execution. As will be demonstrated in Section 7.3.3.4.3, the greater modularity of compilation also plays a fundamental role in lowering the performance cost of compilation relative to interpretation.

- **Inter-project Information Sharing.** The object code modules that result from compilation of each module can be shared directly between projects, and packaged into statically or dynamically sharable libraries. While interpreters typically allow the sharing of source code files between projects, they offer no opportunity for the sharing of information derived from source modules between executions of the interpreter or different projects.

- **Reuse of Information between Analysis Runs.** Looked at from the standpoint of information usage, interpretive mechanisms are extremely wasteful, as they typically discard information gained concerning the structure of the source code and program throughout an execution. By contrast, compilers preserve program metadata between executions and compilations of a program, providing an opportunity to reuse the results of earlier compilations and to use the derived metadata to interface more smoothly with other programming components. The run-time savings that comes from *reuse* of modules once they are compiled is key to the efficiency advantages offered by a compiled approach.

- **Metadata Availability.** Advanced operating environments store metadata concerning the structure of scripting and event interfaces available in compiled code. These interfaces typically are static, remaining as standards once they are established. If we wish to build a componentized analysis system that meshes well with scripting and other extensibility techniques, compilation yields a more natural and simpler solution than does interpreter-based analysis.

- **Potentially Higher Analysis Performance.** The items above mentioned some considerations – such as the modularity of the program and the reuse of program information between execution – that help ameliorate the up-front cost of compilation in compiled analysis systems relative to interpretive approaches. The critical advantage offered by a compiled analysis approach, however, concerns the greater speed with which the compiled analysis itself is conducted. There are two important reasons for this improved performance[36]:

  - **Elimination of Interpretive Overhead.** A programming language interpreter can be applied to *any* legal input program. During its operation, the interpreter both discovers the structure of the program being analyzed and performs the actions entailed by that structure. A given sequence of code is frequently re-interpreted many times over, with the interpreter simplemindedly rediscovering the structure of a that sequence each time. It is well-known that "interpretive overhead" can lead to very significant performance degradation, particularly when the amount of actual work required for each interpreted unit is relatively small.

  - **Efficient Resource Mapping.** Whether it is based on compilation or interpretation, any implementation of a programming language must decide how to map variables and other resources within the program being simulated to resources in the underlying machine. For a compiler, the mapping is typically rather efficient, and the execution of the resulting code can make direct use of machine resources. A few examples of this mapping include

    - Register allocation of frequently used scalar variables.

    - Mapping of program stack frames to efficiently maintained, contiguous blocks of storage, and the use of dedicated stack and frame-related registers for maintaining this storage.

[36] Note again that both of these disadvantages can be ameliorated through the use of *hybrid* analysis methods, and do not require adoption of the full-fledged compiled analysis methodology. As noted earlier, this approach has been investigated neither in this thesis nor elsewhere in the literature, to the best of the author's knowledge.

- Use of built-in instruction pointer and related resources (e.g. delayed execution slots following branches, instructions to maintain the instruction pointer).

- Mapping of heap allocations to efficiently allocated blocks, maintained by system libraries without further program bookkeeping information.

By contrast, a programming language interpreter typically does not have the option of using resources in such an efficient manner. The resources simulating program data within an interpreter are typically accessed through an additional level of indirection (e.g. the lookup mapping variable names to particular locations). Maintaining the resources used by the program in this framework typically requires considerable amounts of bookkeeping and administrative machinery. Both of these patterns of behavior prevent the effective allocation of resources to the program being analyzed, even when that allocation would offer strong efficiency advantages. For example, it is not feasible in an interpreter to make use of the built-in call stack, frame pointer, and saved-away instruction pointers, because the interpreter itself requires the use of such resources in its own operation. The inability of an interpreter to map the underlying resources effectively can result in significantly decreased performance.

7.3.2 Disadvantages of Compilation Frameworks

The previous section discussed some of the advantages offered by the use of compilation in program analysis. This section briefly examines two areas in which compiled analysis is *less* desirable than interpreted analysis.

- **Larger Code Size.** Both compiled and interpreted systems require the representation of a program's code and data while the program is running. An important difference between the two approaches, however, relates to the fact that a typical interpretive analysis system is typically designed specifically to operate on source code in a particular programming language. By contrast, the "interpreter" employed by a compiled analysis system is the microprocessor, which implements a extremely general and low-level language. An efficient encoding of the program for an interpreter will therefore typically require dramatically less space than a compiled version of that program.[37] In certain cases, the

[37] Such space savings is not, however, guaranteed: The relative representational cost of interpretive and compiled analysis systems depends in part on the level of detail associated with the language implemented by the interpreter. For example, a straightforward encoding of a program in a an extremely low-level language (such as the register transfer languages common in intermediate compiler representations), could require space comparable or greater than that required for an encoding of that program in a machine language.

relatively higher space demanded by the compiled representation can cause considerable performance degradation due to memory system effects (e.g. cache or page thrashing).

While interpretive systems may represent program code in a more efficient manner, the opposite may be true of program data (due in part to the efficient mapping of data structures to machine resources). While the ratio of memory consumed by code and data will vary for different programs and different executions of the same program, for many programs the aggregate amount of memory used can considerably outweighs the size of the program executable. Although this can help balance out the greater space required of compiled code, the details of the overall balance depend on details of the encoding techniques used and the behavior of the particular program being discussed.

- **More Involved Program Modification.** As formalized in the first Futamura projection [Futamura 1971], compilation can be conceptualized as the specialization of an interpreter to a particular source program. The specialization typically yields a program whose execution is considerably more efficient but less flexible than the original combination of interpreter and source code. In particular, for an interpretive analysis system, the source code represents simply a structured input to the interpreter, and can therefore be easily modified and reinterpreted without further effort. By contrast, within a compiled analysis system, modifying the original source code requires both recompiling the program (to create the "compiled analysis" program) and then re-executing the program.

7.3.3 Efficiency Concerns

The aggregate time required by interpretive- and compiled- analysis systems will vary widely depending on a number of factors. Under certain circumstances, an interpretive system requires less computational resources, while other circumstances will make a compiled system more desirable. This section examines factors affecting the relative efficiency of interpretive- and compilation-based approaches to program execution. We examine this from two angles: First, we identify the most important contextual factors that combine to determine how frequently the pieces of a program are interpreted and compiled. We then proceed to construct a model of these factors in order to examine how they affect the relative benefit of compilation-based approaches over their natural ranges of variation.

7.3.3.1 Compiled and Interpreted Analysis : Operational Distinctions[NO106]

Before beginning the analysis itself, we briefly present the general framework in which it is assumed that compiled- and interpretive- analysis takes place. These frameworks will then form the basis for the model presented in Section 7.3.3.2.3.

The first type of system we examine is an *interpretive* generic analysis system, capable of interpreting any legal program on any appropriate sequence of user-specified domains. It is assumed that such a system would parse the program of interest, convert it to an appropriate internal form, and connect to the appropriate user-specified domains. The abstract interpreter would then proceed to perform abstraction execution on the program using the user domains. The operation of such a system is schematically depicted in Figure 40.

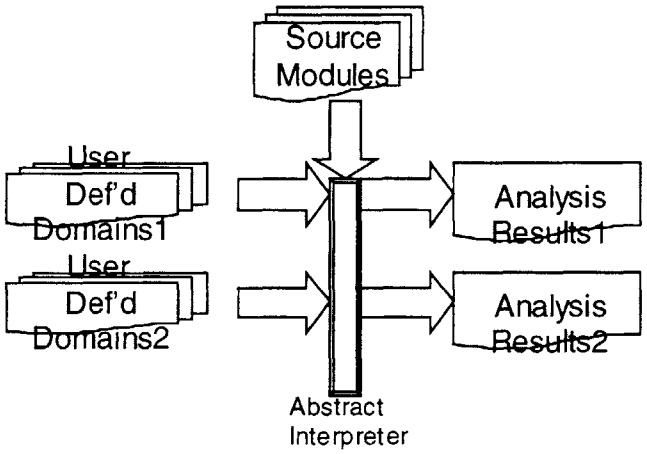

Figure 40: The Operation of a Interpreted Generic Analysis System.

A *compiled* generic analysis involves more steps, as seen in Figure 41. The first step in such a system is the compilation of the user's program to an "analysis executable" that actually performs the interpretive execution. Given the existing structure and mechanisms of programming environments and patterns of user behavior, this compilation is likely to take place incrementally on a module-by-module basis. A given change by the user to the program will typically be rather localized, and will require the compilation of only a small number of modules. Each such compilation for a particular module involves the parsing of the module and emission of the appropriate generic, abstract semantics code. Following an incremental series of such compilations, the compiled modules will be linked together into an analysis executable. At some later points, this analysis executable will then be invoked to perform analyses on the program. The ratio of analyses to compilations is expected to vary strongly for different users. (For a user seeking to understand a pre-written program or computational system, such analyses might take place without any modifications to

the program. By contrast, a developer performing analyses for debugging or testing purposes might frequently make incremental changes to the program and recompile.)[38]

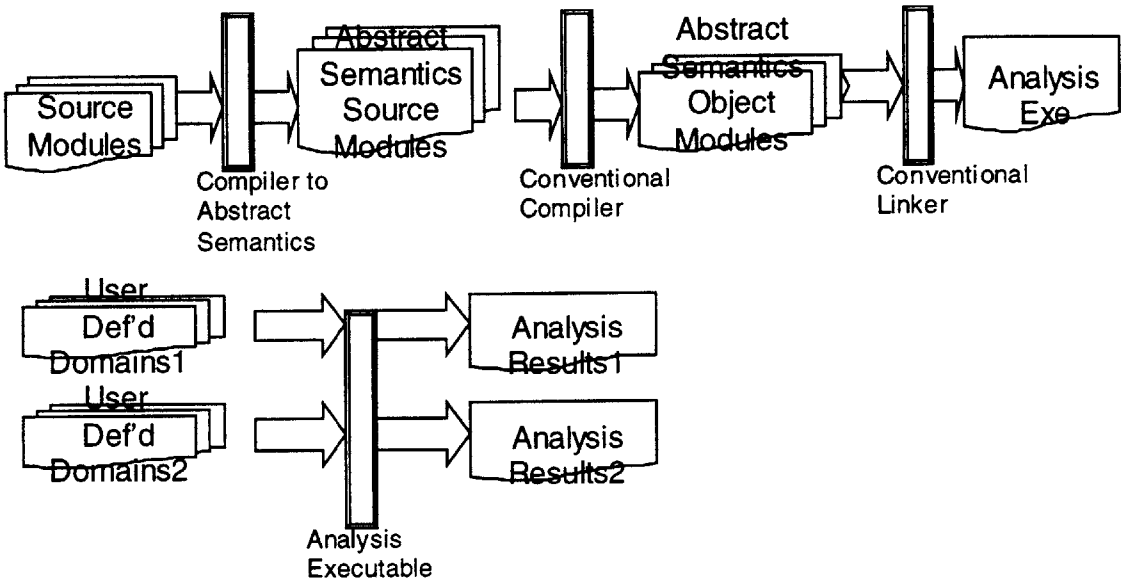

Figure 41: The Operation of a Compiled Generic Analysis System.

The differing patterns of usage dictate many of the most important performance tradeoffs of the compiled and interpretive systems, and form the basis for the model presented below.

7.3.3.2 Efficiency Considerations

Given the operational model laid out in the previous section, we turn to informally survey the primary factors contributing to the performance of the compiled and interpretive analyses. Each of these factors will then be formalized and assume a role in the model presented in the next section. The presentation will start with issues relevant to the computational cost of the interpretive system; factors of relevance to the compiled system will follow.

7.3.3.2.1 Interpretive System

The interpretive system is modeled as reading all modules, translating the modules into an internal format, connecting to the user's semantic domains, and performing an abstract interpretation on the program.[39] A few issues are central to understanding the cost of the interpretive system:

---

[38] Note that while TACHYON currently compiles to *source code*, for the sake of the performance model it is assumed that the compilation takes place directly to object modules. While such an approach would reduce the cost of compilation, it would also have deleterious effects on the capacity of the user to *debug* the abstract semantics program – thus eliminating an advantage of compilation-based systems discussed in Section 7.3.1.

- **Startup Cost.** An invocation of the interpreter will be associated with a certain startup cost arising from process creation, loading the executable into memory, and initialization of data structures.

- **Performance Cost for Parsing and Internalizing Source.** Prior to executing a program, the interpreter must read that program into an internal format. Because an interpreter must read the *entire* program, this parsing can impose significant overhead for larger programs. We would expect this cost to include both a fixed time component and one linear in the size of the program (associated with the parsing and the translation to internal data structures).

- **Efficiency of Abstract Execution.** For each line of code executed during analysis, an interpretive, extensible abstract execution will experience overhead from both the interpretation of constructs and from the late binding of the semantic domains. It is important to recognize that the metric applied here is the cost per "dynamic" line of code executed by the program, rather than the cost per line of code in the program text. (Thus, if a loop body consisting of n lines of code is iterated c times before a fixed point is reached, the cost will be proportional to cn rather than simply to n.)

- **Length of the Analysis Code Sequence.** In order to judge the overall cost of the analysis, we must consider not only the efficiency of the execution, but the length of the execution sequence itself – the number of "dynamic" lines of code executed in a program before abstract execution converges to a fixed point. Because the semantic rules being applied are the same, the paths taken during interpreted and compiled analysis are identical. This parameter will play an important role in the understanding the aggregate time required by the compiled analysis time as well. While there is in general no way to relate the running time of an algorithm in the standard semantics to its code length, we can place rough asymptotic bounds on the running time of *abstract* analyses given an understanding of the heights of user-defined domains. Section 7.4 briefly examines these issues.

Because of the operational simplicity of the interpretive system, relatively few parameters are required to model its performance. Presentation will now turn to the more involved characterization of the compiled analysis model.

7.3.3.2.2 Compiled System

The operation of the compiled system involves two primary stages of operation: Compilation and linking of modules to form a generic analysis executable, and subsequent execution of the analysis executable with a

[39] Note that while these processes may not occur in this order, it is expected that all of them will occur at some point during interpretive execution.

series of user-specific semantic domains. Understanding the efficiency of this process requires consideration of a number of parameters drawn from each of these stages.

- **Number of Analyses performed per Compilation.** The compiled framework requires an extra compilation step beyond that required by the interpretive framework, with the reward being a faster analysis process. A critical factor in judging the relative cost of the compiled and interpretive analysis is thus the specification of the relative frequency with which compilations and analyses take place. If on average a compilation must be performed for every analysis, the compiled analysis is unlikely to offer much or any savings in analysis time. On the other hand, if many analyses are performed for a given compilation, the higher performance of those analyses may more than recoup the cost of the compilation. A specification of the analysis/compilation ratio is thus very important for assessing the overall aggregate time tradeoff.

- **Cost of Compilation Step.** The computational expense associated with the compilation step is key to judging the aggregate efficiency of the compiled analysis, and can vary considerably based on the structure of the program and the characteristics of the user's changes to the program. The following two considerations are particularly key in assessing the cost of compilation.

  - **Cost of Parsing Program and Emitting Specialized Analysis Code for a Module.** When the user updates the code within a module, the compiler to the abstract semantics must re-parse the module and emit new analysis code. Given the inefficient compiler currently implemented for TACHYON, this process includes both a fixed component (e.g. for process creation, executable loading time, and reading in the statement templates) and an algorithm linear in the size of the program. Compilers that are more efficient could significantly reduce the cost of each of these components, but fixed and linear costs are likely to remain.

  - **Number and Sizes of Modules to be Compiled for each Compilation Step.** As noted in the previous section, most updates to a program occur incrementally. Given a modular program, compilation following a modest set of changes will typically be confined to a small set of modules. The overhead associated with a compilation therefore reflects not the cost of compiling the entire program, but only the particular modules that were changed. Unfortunately, it is also true that the modules within a given program vary widely in size. All other things being equal (i.e. if changes are assumed to be distributed evenly among lines of code), changes will tend to cluster in the larger modules. Important considerations thus include both the size distribution of the program modules,

and the induced distribution associated with the total size of the modules affected by program changes. Two of the more important factors in these costs are specified below.

- **Distribution of Source Code Changes per Compilation.** In general, each compilation event will take place after a number of changes to the source code. For a program with many modules, a greater the number of changes will typically lead to the need for processing a greater number of modules in the subsequent compilation step.

- **Distribution of Source Code Changes over Lines of Code.** The modification of lines of code prior to a compilation will take place according to some spatial distribution. If the source code lines are tightly clustered, fewer modules will require compilation. If, on the other hand, the changes are more widely distributed, a program broken into many modules will typically have to recompile many such modules for a given compilation step.

- **Distribution of Module Sizes.** Knowledge of the distribution of *module sizes* is important both in terms of how it impacts the distribution of changes over modules and in terms of how it affects the mean module compilation time. As noted above, the random distribution of source code changes among lines of code tends to probabilistically skew changes towards occurring in larger modules, since those modules span a larger set of lines of code. In addition, the presence of a wide variation in module sizes can lead to a large variation in associated compilation times. An important element of the model is thus understanding the distribution of modules sizes – if it exhibits large variance, compilation times may be compilation of relatively few large modules, rather than a larger number of smaller modules.

- **Cost of an Analysis Step.** By contrast with the somewhat involved characterization of the compilation times, the considerations involved in understanding the speed of the compiled analysis are rather simple. In particular, there are three major contributions to analysis cost::

  - **Administrative Overhead.** Each compiled analysis will incur some amount of overhead prior to, during, and after the analysis that does not directly stem from the analysis itself. For example, time is required to create the analysis process, load code from persistent storage, initialize analysis data structures, etc. As was the case for the analogous overhead for the interpretive system, this startup time can be characterized as a fixed cost plus a cost linear in the size of the program being loaded.

  - **Efficiency of Abstract Execution.** The primary goal of adopting a compiled analysis strategy is the capacity to conduct analysis that is more efficient. The higher performance of the compiled

analysis stemming from the mapping of analysis structures to machine resources and the elimination of interpretive overhead is thus central to an understanding of the tradeoff between interpreted and compiled analyses. We will measure the efficiency of abstract interpretation in terms of the cost per dynamic lines of code executed.

- **Length of the Analysis Code Sequence.** As was discussed in the context of interpretative analysis, in order to judge aggregate analysis cost it is necessary to consider not only the efficiency of the abstract execution but the length of the execution trace over which the execution is performed. We will measure this trace in terms of dynamic lines of code.

7.3.3.2.3 Conclusion

We have noted the most critical factors affecting the aggregate, amortized time consumed by compiled and interpreted analysis systems. This discussion has left us with a sizeable number of factors, some of which interact in rather subtle ways. The large variation in program structure and in patterns of human interaction with programming systems suggests that we need to examine the compilation/interpretation tradeoff over a substantial volume of parameter space. In order to address both of these reasons, we turn now to formulate a mathematical model of the issue. We then instantiate the model with representative parameters, examine the sensitivity of the model to parameter variation, and consider the tradeoffs associated with differing program characteristics and regimes of user interaction.

7.3.3.3 A Mathematical Model of Aggregate Analysis Performance

This section formalizes the intuitions presented above, by placing the most important parameters affecting the tradeoff between compiled and interpretive analysis into a probabilistic mathematical model. We will first examine the general form of the model, and will then characterize model costs as combinations of samples drawn from a number of fundamental distributions.

We model the ratio of the aggregate times required by a compiled analysis to the aggregate time required by the interpreted analysis as a random variable. This random variable is a function of 10 parameters (some of which are themselves tuples of several related parameters) and is associated with a derived distribution that is expressible as a combination of several other, more basic, random variables:

$$\rho_{\gamma, h, m, \mu, \sigma, n, \vec{I}, \vec{C}, \vec{T}, \vec{A}}(\rho_0)$$

The variables parameterizing the model are as follows:
γ : Compiles/Analysis Run. The mean number of compiles per analysis run.
h : Changes/Compile. The mean number of 1-line source changes per compile.

n: Unit. Count of modules in the system

μ : Lines of code. The mean size of a module.

σ: Lines of code. The standard deviation of the module size.

$\vec{I}$: Three performance parameters dictating the running time of the interpreter:

α_i : Clock cycles. The fixed cost component of invoking and executing the interpreter.

β_i : Clock cycles/Source Line of code. The cost term of the interpreter[NO107] linear in terms of the length of the program.

λ_i : Clock cycles/Executed Line of code. The cost term of the interpreter[NO108] linear in terms of the length of the analysis.

$\vec{C}$: Two performance parameters dictating the running time of the compiler to the abstract semantics:

α_c : Clock cycles. The fixed cost associated with the compilation of a module to the abstract semantics.

β_c : Clock cycles/Source Line of code. The linear cost associated with the compilation of a module to the abstract semantics.

$\vec{A}$: Three performance parameters dictating the running time of the compiled analysis code.

α_a : Clock cycles. The fixed cost associated with invoking and executing the analysis executable.

β_a : Clock cycles/Source Line of code. The cost associated with invoking and executing the analysis executable linear in the length of the executable.

λ_a : Clock cycles/Executed Line of code. The cost term of the analysis executable linear in terms of the length of the analysis.

$\vec{T}$: Four performance parameters dictating the length of the analysis (executed lines of code) in terms of the source code length in lines of code.

t_0 : Unit. The coefficient for the linear term of the analysis running time.

t_1 : Unit. The coefficient for the linear term of the analysis running time.

t_2 : Unit. The coefficient for the quadratic term of the analysis running time.

t_3 : Unit. The coefficient for the cubic term of the analysis running time.

We further approximate the distribution of changes per compilation as a Bernoulli distribution with $p = 1-(1/h)$, and characterize the distribution of module sizes as a normal distribution with mean μ and standard deviation σ.[40] The length of the executable image is assumed proportional to the length of the original source code.

7.3.3.3.1 Interpretive Analysis

The cost of an interpretive analysis is modeled as a derived random variable $I(i_0)$ based on the random variable $m(m_0)$.

[40] Note that this distribution would be better modeled by a higher-order Erlang or Bernoulli distribution, as the normal distribution assigns non-zero probability to the unphysical possibility of negative-size modules.

Given $M = \{m_1, m_2, m_3 ... m_n\}$, where each $m_i \in \aleph$ is an independent sample from $m_{\mu,\sigma}(m_0)$ and represents the size of a source module in lines of code.

Let $|M| = \sum_{i=1}^{n} m_i$.

$$I(i_0) = \alpha_i + \beta_i |M| + \lambda_i \left(t_0 + t_1 |M| + t_1 |M|^2 + t_3 |M|^3 \right)$$

7.3.3.3.2 Compiled Analysis

As discussed in Section 7.3.3.2, the compiled analysis approach sustains two sets of costs: The cost associated with compiling the program to an analysis executable, and the cost of invoking and running that executable. The two component costs are examined in the following sections, and the amortized aggregate cost is presented in Section 7.3.3.3.2.3.

7.3.3.3.2.1 COMPILATION TO AN ANALYSIS EXECUTABLE

In modern programming systems, compilation takes place incrementally, with each compilation event being preceded and caused by a set changes to lines of code by the programmer. We model this process as consisting of a set of *c* changes, where *c* is a random variable drawn from a Bernoulli distribution with mean *h*. In perhaps the most unlikely aspect of the model, these changes are modeled as taking place to independently distributed lines of code. The location of each of the changes is randomly selected from a uniform distribution over the set of all lines of code in the modules. The mapping of these modified lines of code to their associated modules then provides the of modules that must be compiled in the compilation step. The cost $C_C(c_0)$ associated with the compilation is a random variable given by the following:

Let N represent the set of all modified modules.

$$N = \bigcup_{i=1}^{c} \left\{ m_j \left| \sum_{k=1}^{j-1} m_k \le x_i < \sum_{k=1}^{j} m_k \right. \right\}$$

Where

c is a random variable representing the number of changes, and is drawn from the Bernoulli distribution $b_h(b_0)$ with mean h.

Each x_i is a random variable drawn from the uniform distribution between 1 and $|M|$.

Then the time required for compilation of the modules to the abstract semantics can be given by the random variable

$$C_C(c_0) = \alpha_c + \beta_c \sum_{i=1}^{|N|} N_i$$

7.3.3.3.2.2 EXECUTION OF THE ANALYSIS EXECUTABLE

The second component of the aggregate cost of the compiled analysis approach consists of the time required by the execution of the analysis executable. Like the execution of the interpretive system discussed above, the execution of the analysis executable contains three components: A fixed component, a component proportional to the length of the (in this case, compiled) program, and a component proportional to the length of the analysis path (representing, in essence, the *speed* with which execution takes place). The length of the analysis executable is itself a random variable involving both fixed, linear, quadratic, and cubic components. The t_i specify the coefficients associated with each of these terms. The cost of the execution process can thus be expressed as a random variable $C_A(c_0)$:

$$C_A(c_0) = \alpha_a + \beta_a|M| + \lambda_a\left(t_0 + t_1|M| + t_2|M|^2 + t_3|M|^3\right)$$

7.3.3.3.2.3 AGGREGATE COST

The two sections above have formulated expressions for the random variables associated with the cost of compilation and the cost of running the abstract analysis. This section simply combines those results in the appropriate proportions. Recall that the ratio of the frequency of compilations to analysis executions is given by the parameter γ. Given the results from above, the aggregate cost of the compiled analysis per particular analysis can thus be expressed as $C(c_0)$:

$$C(c_0) = \gamma C_C(c_0) + C_A(c_0)$$

$$C(c_0) = \gamma\left(\alpha_c + \beta_c \sum_{i=1}^{|N|} N_i\right) + \alpha_a + \beta_a|M| + \lambda_a\left(t_0 + t_1|M| + t_2|M|^2 + t_3|M|^3\right)$$

7.3.3.3.3 Ratio

Pulling together the results of Sections 7.3.3.3.1 and 7.3.3.3.2.3, we can characterize the random variable $\rho_{\gamma,h,m,\mu,\sigma,n,\bar{I},\bar{C},\bar{T},\bar{A}}(\rho_0)$.

Let $m_{\mu,\sigma}(m_0)$ be a normally distributed random variable with mean μ and standard deviation σ, representing the distribution of module sizes.

Let $M = \{m_1,...m_n\}$, where each $m_i \in Z$ in $\{m_1,...m_n\}$ is an independent sample from $m(m_0)$, representing the size of a module.

Let N represent the set of all modified modules, given as $N = \bigcup_{i=1}^{c} \left\{ m_j \mid \sum_{k=1}^{j-1} m_k \le x_i < \sum_{k=1}^{j} m_k \right\}$

Where c is the a random variable representing the number of changes, and is drawn from the Bernoulli distribution $b_h(b_0)$ with mean h, and each x_i is a random variable drawn from the uniform distribution in the interval $[1, |M|]$

The ratio of aggregate user time consumed by the compiled compared to the interpreted analysis is given by the random variable

$$\rho_{\gamma, h, m, \mu, \sigma, n, \vec{I}, \vec{C}, \vec{T}, \vec{A}}(\rho_0) = \frac{C(c_0)}{I(i_0)}$$

$$= \frac{\gamma\left(\alpha_c + \beta_c \sum_{i=1}^{|N|} N_i\right) + \alpha_a + \beta_a |M| + \lambda_a \left(t_0 + t_1 |M| + t_2 |M|^2 + t_3 |M|^3\right)}{\alpha_i + \beta_i \sum_{i=1}^{n} m_i + \lambda_i \left(t_0 + t_1 |M| + t_1 |M|^2 + t_3 |M|^3\right)}$$

7.3.3.4 A Monte Carlo Simulation of the Mathematical Model

A Monte Carlo simulation was created from the mathematical model formulated above in order to empirically investigate the relative costs between compiled and interpretive analysis. Samples of model outcome were take by randomly formulating source code and source change profiles drawn from the appropriate probability distributions and testing the relative analysis times using the formulas given above.

The high dimensionality of $\rho_{\gamma, h, m, \mu, \sigma, n, \vec{I}, \vec{C}, \vec{T}, \vec{A}}(\rho_0)$ made it infeasible to simultaneously observe variation across the full sample space. The investigation therefore first assessed the sensitivity of the model around a plausible set of parameter settings. We then independently examined in detail the variation along those dimensions with the greatest impact on the ratio ρ or which otherwise seemed to be of particular interest. The Monte Carlo simulation was performed in the context of a Microsoft Excel spreadsheet with the simulation performed by Visual Basic code at the URL given in Appendix 4. The major issues examined are presented in the sections below.

7.3.3.4.1 Default Parameter Settings

In order to supply default values for the examination of variation along different dimensions, empirically measured or plausible values were assigned for different model parameters discussed above. Table 28 shows the default settings of the parameters, and notes the qualitative sensitive associated with each. Certain of the

parameters (such as the ratio of compilations to analyses, γ, and the mean number of changes per compilation, *h*) are expected to vary widely in practice, and are thus examined below. For other parameters, investigations below are motivated by a desire to understand of the qualitative effect of that parameter on the overall system (compiled- or interpretive- analysis) performance, or because that the system exhibits great sensitivity to that parameter. Note also that some of the most crucial values -- those dictating the length of the analysis process -- depend on the particular user-defined domains used within the analysis, and thus cannot be given even precise emprical values.

Parameter		Default Value	Sensitivity	Comments
	γ	.5	High	Depends on use patterns; high variation possible.
	h	2	High	Also depends heavily on patterns of use, level of programming interactivity.
	n	200	Medium	Programs will vary widely in # of modules.
	μ σ	100 50	Medium Low	Programs will vary widely in both mean size and variation of module sizes.
	α_i β_i λ_i	400000000 13333.3 600	High Medium Low	These depend on the structure of the interpretive analysis.
	α_c β_c	400000000 20000	Low High	These depend on the engineering of the compiler to the abstract semantics.
	α_a β_a λ_a	80000000 0 150	High Medium High	These parameters depend on the code emitted from the compiler to the abstract semantics. The efficiency per executed line of code (λ_a) is of particular interest.
	t_0 t_1 t_2 t_3	0 1 0.15 0	High Low	These parameters are very rough approximations, and the real-life numbers will differ widely depending on the particular user domains in place..

Table 28: Default Values for Parameters and their Sensitivity at that Point.

7.3.3.4.2 *Fundamental Factors: Length of Analysis, Granularity of Program, and Ratio of Changes to Analysis*

Aside from the relative efficiency of the compiled and interpreted code, investigation identified two parameters as worthy of particular note:

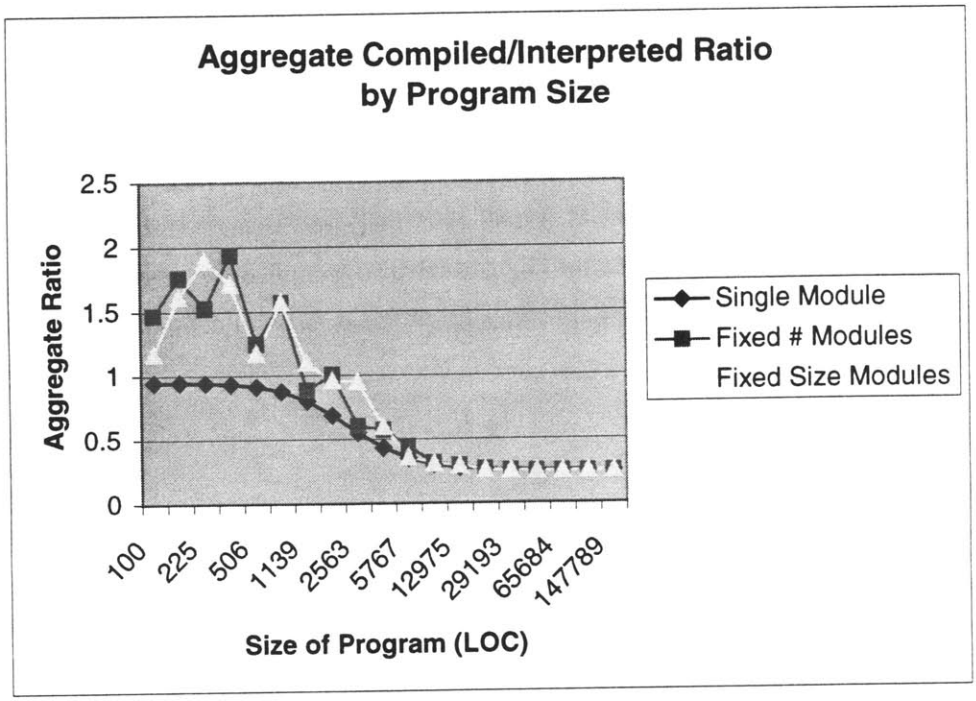

Figure 42: Variation in the Attractiveness of Compiled Analysis with the Program Size.

Length of the Analysis. The longer the analysis path that must be analyzed, the greater the fraction of time that the interpreter and compiler will spend in the process of analysis. As a result, the aggregate ratio of the performance of the compiled vs. the interpretive system will increasingly approximate the ratio of the associated analysis speeds, rather than being dominated by other factors such as the time for compilation, or the parsing time of the source code. Note that the analysis path length can be increased either by the processing of a larger program, or by the use of analysis algorithms exhibiting slower convergence time. Figure 42 illustrates how the ratio of aggregate analysis time approaches the ratio of analysis performances as the size of the program increases. The details of this approach do differ, however, for programs whose modules exhibit different degrees of granularity. As the program grows longer, the analysis path length will increase, and the higher efficiency of a compiled analysis approach will overcome losses due to compilation time. Notice the "noise" in the measurements associated with more than one module, arising from stochastic variation in the number of modules that need to be compiled in each compilation event – a significant cost for smaller programs. This noise obscures, but does not eliminate, the evidence of a "concave downwards" curve for smaller programs – when multiple compilations may be involved, there is some program size for which interpretive analysis yields the best results.

Ratio of Compilations to Analyses. A compiled-analysis strategy bears additional costs each time a compile takes place, but reduces the time the user spends in analysis. It thus stands to reason that the ratio of aggregate times will vary with the ratio of compilations to analyses. The Compilation/Analysis ratio will typically differ substantially in different contexts. For example, a developer may run compilations relatively frequently when debugging a program, while testing runs may be much more heavily biased towards analyses. End-user analyses of dynamic systems might involve no compilations at all. Figure 43 shows how the aggregate compiled/interpreted analysis ratio varies for different ratios of compilation events to analysis events.

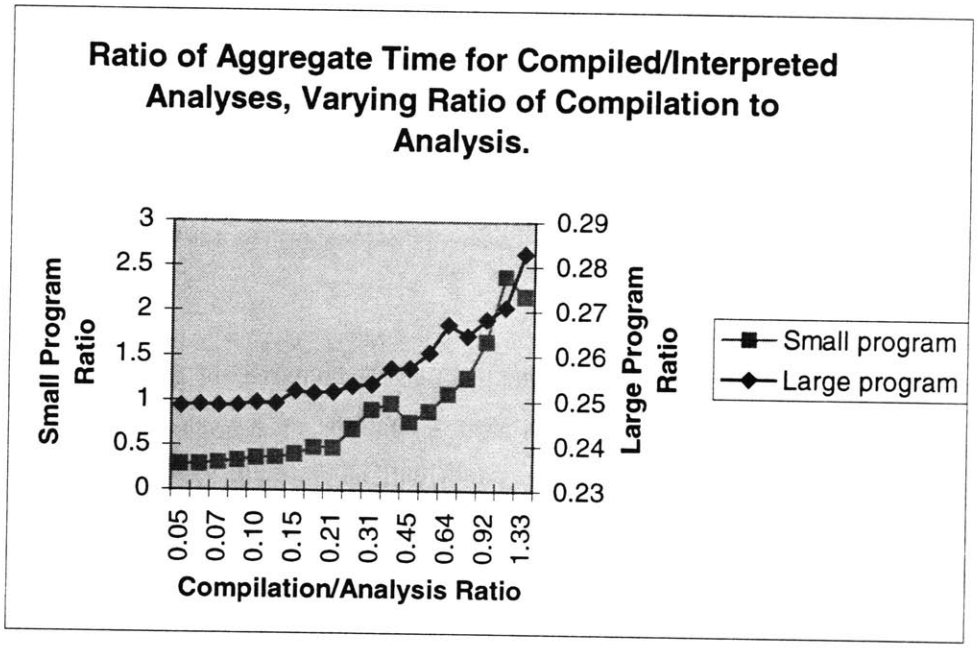

Figure 43: The Impact of the Relative Frequencies of Compilation and Analysis on the Aggregate Time Ratio for Compiled/Interpreted Analysis.

Having identified some of the key issues influencing the aggregate time spent in compiled and interpreted analysis, we turn now to briefly sketch how some other factors impact the relative advantages of compiled analysis.

7.3.3.4.3 Variation in Source Granularity and in Spread of Source Granularity

The compiled analysis strategy always requires a separate step between program changes and final execution. The cost of that step, however, can vary widely, depending as it does on the number and size of

the modules that must be compiled from source code. In a large program, the analysis time advantage associated with compiled analysis is likely to dominate the aggregate running, and the impact of this compilation step will likely be small. (See Figure 44). By contrast, the performance of the compilation step can be extremely important for medium and small sized programs. Perhaps surprisingly, the optimal degree of aggregation in modules is strongly influenced by the number of source line changes that take place per analysis. Figure 44 illustrates how the aggregate compiled/interpreted time ratio changes for a fixed, moderately sized program broken up into different numbers of modules. The change in ratio is examined for two different scenarios: When only a single independent line change takes place per compilation vs. when numerous (6, in this case) changes are batched into a single compilation event. As shown by Figure 44, many, smaller modules are preferable when the compiles take place after few changes, while fewer, larger modules are desirable when a particular compile represents many changes. The explanation for this is not difficult.

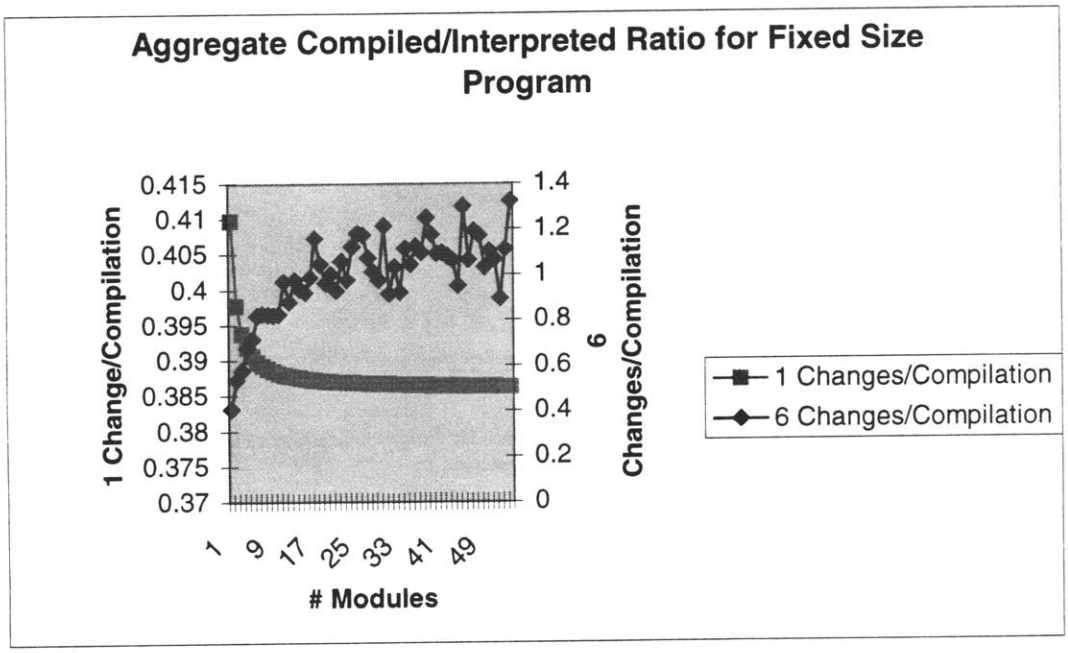

Figure 44: Analysis Ratio for Program with Different Levels of Granularity.

In the case where a compilation takes place after very few changes, there is an incentive to make the modules as small as possible: If a large program has very few modules, the compiler will likely have to compile a very large module even though the few changes are rather localized in the source code. On the other hand, in a program with many modules, the compiler (and incremental linker) will be able to process much smaller fraction of the program – just the module affected by the change.

By contrast, for situations in which a compile takes place after *many* changes, how the changes are distributed will depend on the level of aggregation of the program. If the program has few, large modules, many independent changes may lie in the same module, and few modules will require compilation. The substantial fixed overhead of the compiler will therefore only have to be sustained a few times. On the other hand, for a highly disaggregated program containing many small modules, the changes will be spread over many modules and the compiler may have to be invoked several times. The concave down line of Figure 44 shows this phenomenon. (Note that the samples on this curve experiences much more noise due to sampling variation resulting from drawing of the number of changes performed per compile from a Bernoulli distribution. In the case of the other curve, the system always performs a single source change.)

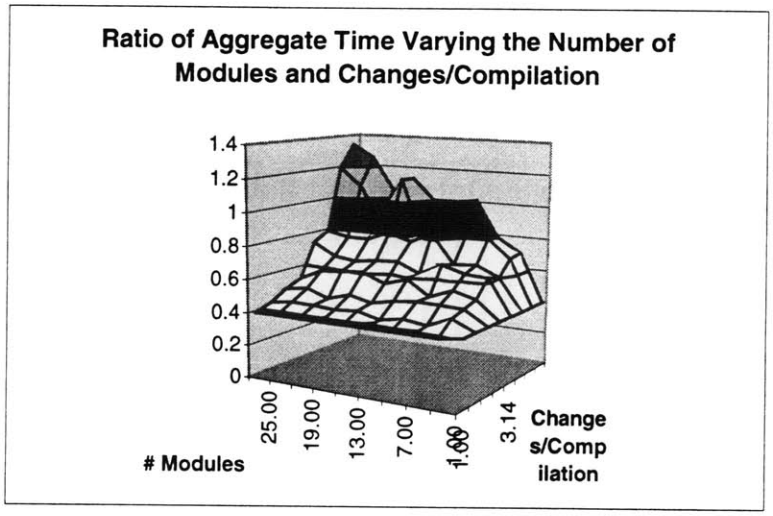

Figure 45: Relative Analysis Performance By Program Granularity and Changes/Compilation.

Figure 45 shows a more global view of how the ratio of compiled time to interpreted time varies for a program that is of fixed size but divided into a varying number of modules and with differing numbers of changes batched into a single compilation. The ideal degree of modularization from the point of view of a compiled analysis depends on the running time characteristics of the compilation as well as the distribution of source code changes that precede a compilation. As the number of changes that take place per compilation event increases, larger modules become desirable.

While one might expect a larger variance in the source module size to impact the relative advantages of a compiled analysis through the probabilistic clustering of changes in larger modules, Figure 46 demonstrates that the running time of the program is relatively insensitive to the variance in the module size.

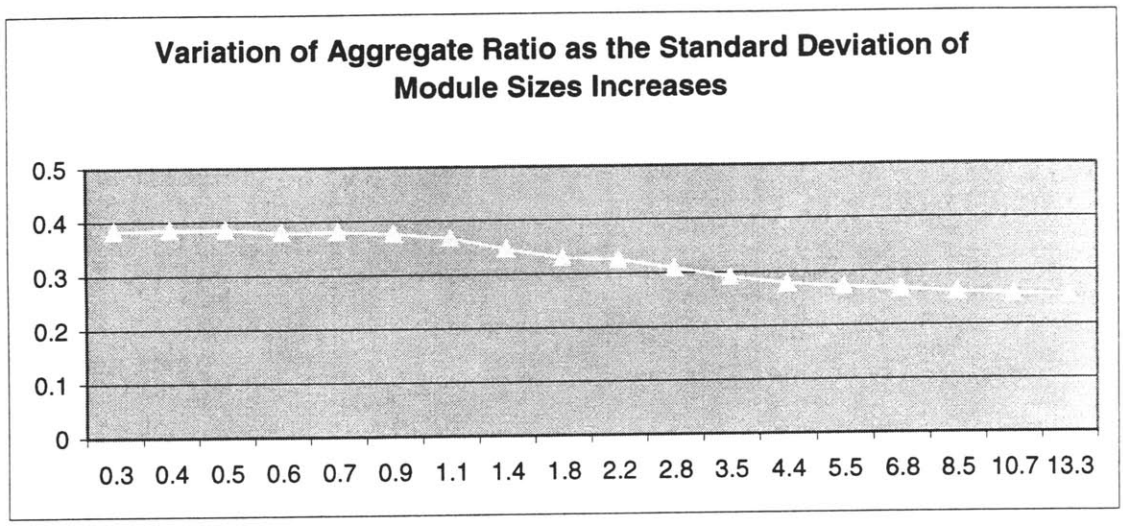

Figure 46: Variation of Aggregate Analysis Ratio with Module Sizes.

7.3.3.4.4 Variation in Total Size of Program

As noted above, the total length of the program is one of the prime contributors in determining the length of the analysis. The length can in turn increase or decrease the relative importance of efficiency in analysis vs. the costs associated with the compiler or interpreter mechanisms. Figure 42 demonstrates how the aggregate compilation/interpretation ratio changes according to the size of the program.

Figure 47 examines this tradeoff for a variety of levels of program granularity as well. In both cases, note that as the program grows, the relative efficiency of the compiled and interpreted analysis increasingly dominates the aggregate performance of the compiled and interpretive techniques.

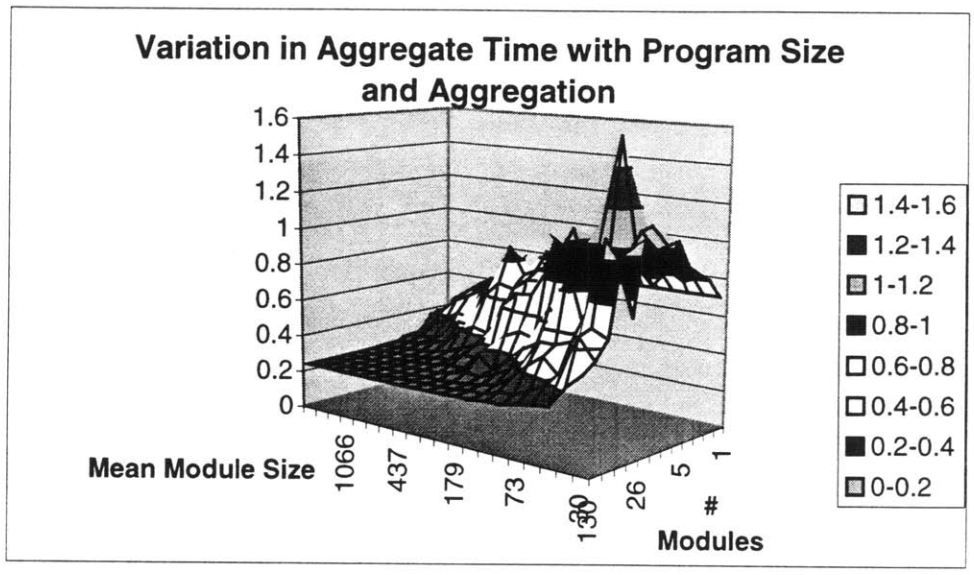

Figure 47: Aggregate Analysis Ratio for Different Sizes and Granularities of Programs.

7.3.3.4.5 *Variation in Order of Analysis*

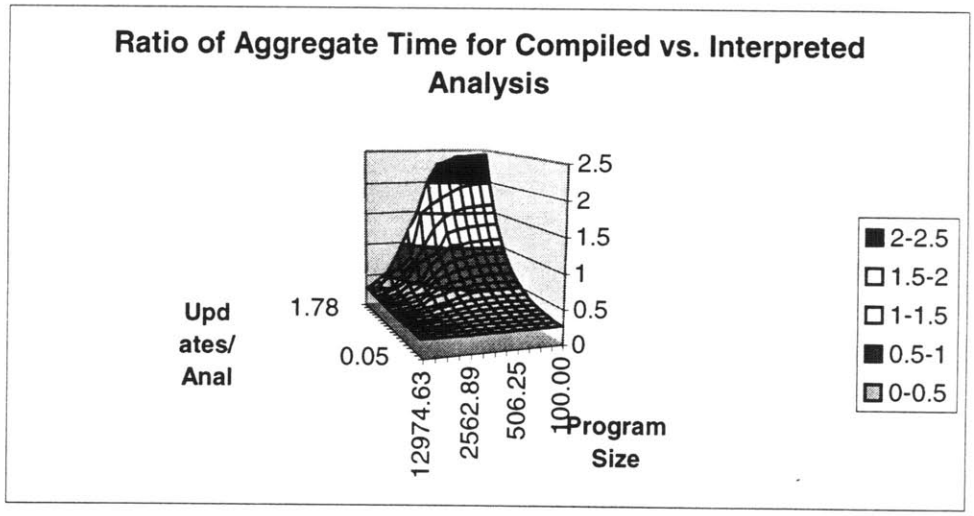

Figure 48: Aggregate Analysis Ratio for Default Analysis Length Assumptions.

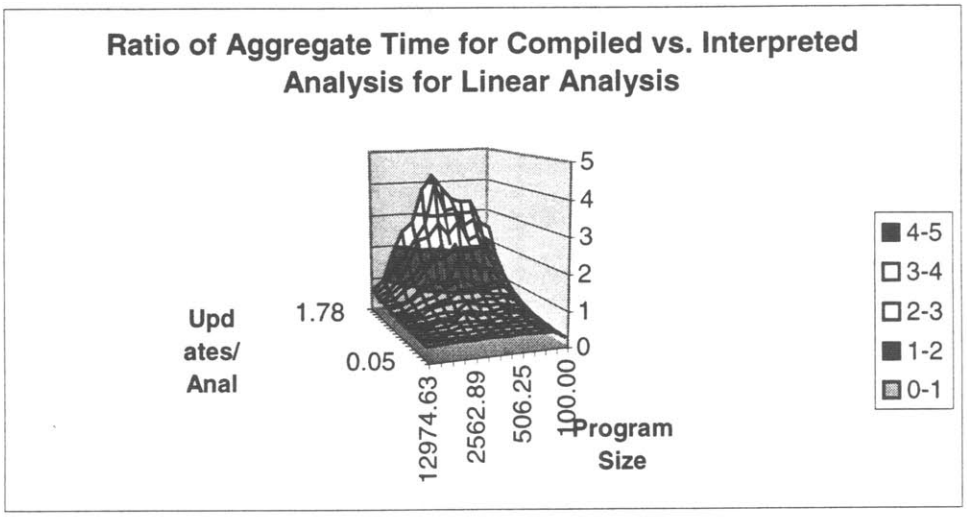

Figure 49: Aggregate Analysis Ratio for Purely Linear Analysis.

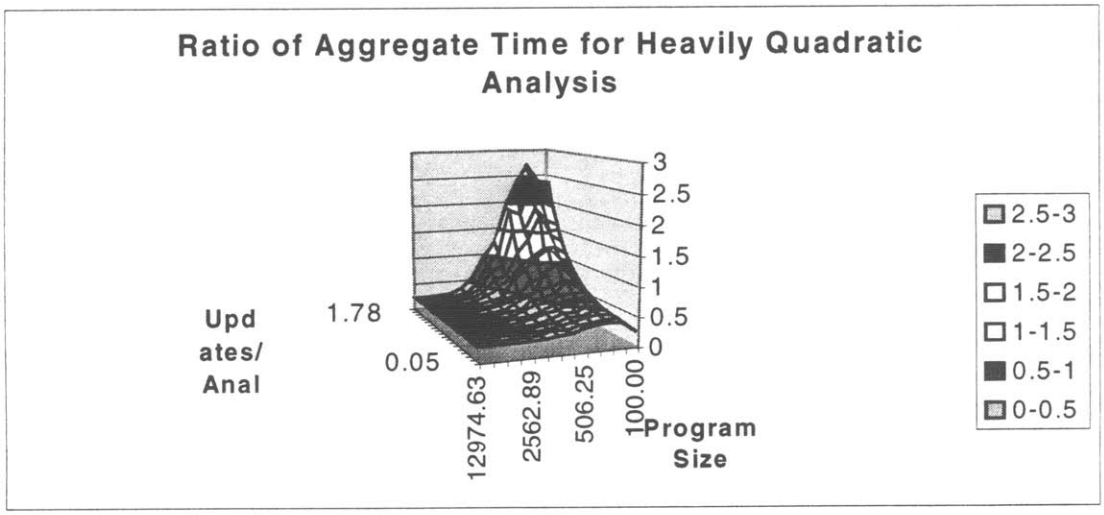

Figure 50: Aggregate Analysis Ratio for Highly Quadratic Analysis.

As discussed in Section 7.4, it can be difficult to judge *a priori* the realistic performance of analysis algorithms such as the used in this thesis. While the order of the performance can be very slow ($o(n^3)$ in worst-case scenarios), it is expected that average-case analysis performance will be much better – quadratic or heavily linear.

Lacking the empirical data to precisely model the expected analysis order for the analysis, we researched the tradeoffs between compiled and interpreted analysis for a variety of asymptotic analysis behavior. Under default assumptions of the length of analysis (that analysis is not purely linear, but mildly quadratic), compiled analysis offers advantages only for those cases in which there is a large program or where the frequency of compiles is significantly below the frequency of analyses. Figure 48 illustrates this case. Figure 49 shows the variation in program behavior for the case of a purely linear analysis. Here, interpretive analysis retains advantages even for large programs as the ratio of compilations to analyses increases. When analyses are much more frequent than compilations, however, compiled analysis retains a significant edge for any size program. Note that the under such assumptions, interpreted analysis can be a factor of two more efficient over the parameter space examined than was the case for the default assumptions shown in Figure 48. Finally, Figure 50 characterizes the aggregate times required by compiled and interpreted analysis when it is assumed that the analysis convergence time will be *heavily quadratic* in the size of the program. The graph demonstrates that compiled analyses quickly becomes very favorable as a program's size grows – regardless of the ratio of compiles to analyses. Similarly, as analyses become relatively more frequent, compiled analysis quickly becomes favorable, regardless of program size.

7.3.3.4.6 Variation in Analysis Speed

As might be naïvely expected, the most critical factor in the ratio between the compiled and interpretive analysis times was the relative speed of the analysis itself. Figure 51 shows how the aggregate amortized time required by compiled and interpreted analysis varies with the relative efficiency of analysis. In order to minimize noise arising from sampling error, the measurements are made for a single-module program. The characteristics of the two curves demonstrate the differential impact of analysis efficiency on large and small programs. In a smaller program, compile time and the overheads associated with starting the compiled and interpreted analyses will tend to play an important role in shaping the aggregate ratio no matter what the relative efficiency of compiled and interpreted analysis. Thus, the variation in aggregate times as the analysis efficiency changes for a small program will be relatively smaller than for a large program, where the analysis time strongly dominates all other considerations in determining the amortized ratio of times. For a large program, the overheads associated with the programming environment are dwarfed by those associated with the analysis itself. As the efficiency of a compiled analysis grows, the aggregate amortized ratio of times is therefore lowered in a much more pronounced manner than for a small program.

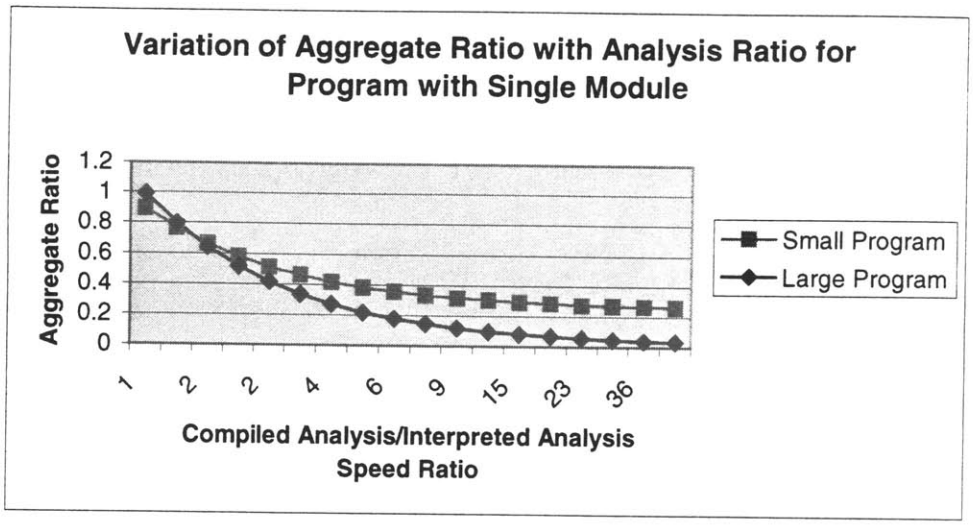

Figure 51: Aggregate Analysis Ratio for With Changes in the Relative Analysis Efficiency.

In summary, the simple model discussed here suggests that compiling an analysis can offer significant improvements in analysis performance, particularly for larger projects. It should be emphasized that this model presents a rather *conservative* approximation to the advantages offered by compiled analysis over interpreted analysis. Empirical data or a more realistic model are very likely to show more significant gains via compilation than are seen here, due to larger analysis times associated with higher linear coefficients on the algorithm's runtime.

7.4 Asymptotic Running Time

This chapter has spent much space describing a technique for boosting the performance of analysis by a constant factor. An important issue not discussed earlier in this thesis concerns the asymptotic running time of the analysis. Here, the approach we have taken offers little flexibility. The important parameters determining the asymptotic running time have been set in previous sections:

- **Straight-line Execution.** In this case, the abstract execution paths are straightforward abstractions from the standard semantics execution paths. Aside from the effects of fixed-point approximation to program loops, a sequence of code of length O(n) will yield an abstract execution path of length O(n).

[41] It is important to be clear that while we are comparing compiled analysis to interpreted analyses, both sets of analyses are modeled as taking place either during or following a step in which the program itself is compiled. While it is possible to envision techniques that would perform analysis on a program in an interpretive manner directly (i.e. without the framework of a compiler in which the analysis takes place or which formulates the bookkeeping structures to be used in for later analysis), such methods are very rare in practice for behavioral analysis, as they require their own language processing framework.

- **Enforcement of O(n) Locations in State.** As described in Chapter 6, in order to ensure convergence of fixed-points the system maps heap allocations into an equivalence class dictated by the point of allocation in the program. For a program of size n, there are O(n) such allocation points. The set of locations associated with variables constitutes another set of size O(n), leading to O(n) total locations in which values are stored.

As discussed further below, the true asymptotic running time of the analysis depends on the choice of abstract state and value domains. Nevertheless, the two analysis features noted above provide a strong constraint in that they guarantee that the lowest asymptotic worst-case running time that can be achieved is $\omega(n^2)$. To see this, consider the fixed-point analysis of a loop. (Depicted in Figure 52) The loop can be of size $\omega(n)$, and thus each iteration of that loop during a fixed point may take time $\omega(n)$. Suppose further that $\omega(n)$ variables are changing within the loop, where each variable is in a nested conditional that prevents its analysis until a previous variable has been abstract to the $\top$ element in the approximating domain. Even given a fixed-height variable domain of height h, the analysis must iterate the loop a number of times proportional to the number of variables, such that each variable is abstracted to the $\top$ element in turn. Because there are $\omega(n)$ variables, $\omega(hn)=\omega(n)$ iterations are required. Because the loop is of size $\omega(n)$ and because the execution must potentially analyze the entire loop body in each iteration, each loop iteration requires time $\omega(n)$. Thus, even for a value domain structure most conducive to fast convergence, the fixed-point iteration must take time $\omega(n^2)$.

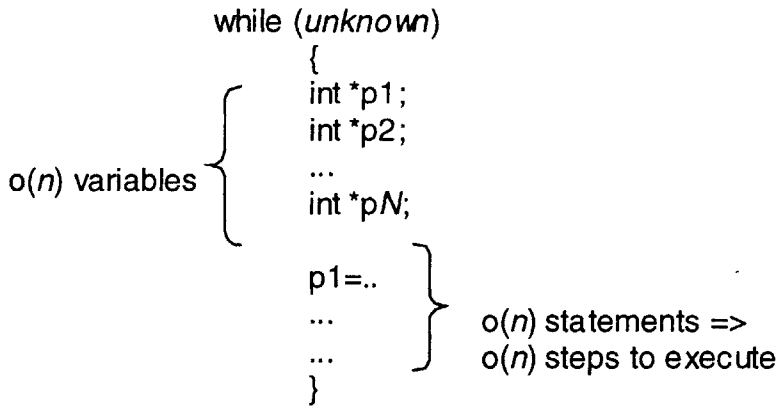

Figure 52: Example of a loop that can take $o(n^2 h)$ time to Converge.

Having bounded the worst-case running time from below, it should be noted that user-specified domain structure can have a large impact on the worst-case considerations. In particular, the maximum height of the user value domains is multiplied by the number of locations and the number of iterations to yield a worst-case estimate of running time. Thus, a user domain of height $\omega(n)$ will yield an aggregate state domain height of $\omega(n^2)$ and a running time of $\omega(n^3)$. Section 12.3 describes an otherwise attractive pointer value domain that yields an $\omega(n^3)$ running time.

7.5 Conclusion

Recent abstract interpretation algorithms have made good use of specialized knowledge of the specific semantic domains in the design of efficient worklist algorithms. The central aim of this thesis is the formulation of a simple route to semantic flexibility, but the design regains some of the lost performance by specializing the analysis to the particular program to be analyzed. While this approach does incur the performance cost of a compilation step, the model presented indicates that compilation can yield strong benefits for a well-structured and sizeable program. Having briefly introduced the motivation and basis for the use of compiled analysis, we now turn to a discussion of the implementation of this approach.

Chapter 8 Compiled Analysis: Engineering

8.1 Introduction

The last chapter examined the conceptual foundation for the compiled analysis approach. That chapter discussed the theory of a specialized analysis, the tradeoffs involved in adopting a between compiled and interpreted analysis approaches, and some discussion of the worst-case asymptotic running times that might be associated with abstract execution. This chapter focuses on the practical aspects of the converting a program to its abstract semantics analogue. There are two compilers currently implemented as part of TACHYON: A compiler for the toy imperative language described in Chapter 4, and a much smaller compiler for system dynamics models written in the IThink™ modeling language (discussed in the introduction). The large majority of this chapter is be devoted to discussion of the first compiler. A section of the end of the chapter discusses the ways in which the model compiler differs from its larger sibling.

8.2 Software Modeling: States

The interfaces associated with program states were presented in Chapter 6. The discussion here will focus on the manner in which the components implementing these interfaces interact with the analysis code. As was discussed in Chapter 4, The handling of abstract states differs significantly from that of abstract values in that not all pieces of the abstract state are encapsulated within a single component. In particular, while an abstract value is a self-contained component, an abstract state is divided into two pieces: A single component implementing the *AbstractState* interface, and a set of variables of type *AbstractValue* that serve as approximations to most program variables.[42] The advantages of this division of an abstract state were discussed in Chapter 4.

The separation of the handling of program variables and other allocated areas has consequences for the design of the value interfaces. In particular, in order to allow variables the same sort of semantic flexibility afforded to normal memory locations, the system requires that each type of abstract state operation represented in the *AbstractState* interfaces be associated with a value domain counterpart

As was discussed in the previous chapter, the modeling with both the *AbstractState* object and the variables are similar in that both maintain distinguished "current" instances. The current instances represent the

[42] "Escaped variables" (variables to which an "address of" operator is applied in the program text) constitute an important exception to this separate handling: In order to permit access to escaped variables through pointer mechanisms, such variables are instead modeled using the AbstractState object rather than as distinct variables in the translated program. Examples of this use of both escaped and non-escaped variables are given in the sections below.

approximation to the abstract state at the point of execution. Patterns in control flow are mirrored by operations on these current instances, and sometimes require the instantiation of a new set of current instances. Table 29 shows how the different control flow operations are handled by both the *AbstractState* component and by the abstract variables.

Operation	**Handling for AbstractState**	**Handling for Variables**
Control flow split	DynamicSplit	GLBWithTruePredicate method
Control flow forward join	LUBToCurrentState	LUB method
Control flow backwards join	FiniteHeightLUBFromCurrentStateAndSetAsCurrentState	FiniteHeightLUB method
Restart Saved State	SetAsCurrentState	Set current copies of variables to saved-away copies of variables.
Create Initial State	CreateInitialState	CreateUninitialized method
Bottom element	Throw bottom exception	No action.
Variable created	CreateVariableLValue	Variable declared
Variable destroyed	DestroyVariableLValue	Variable passes out of scope
Reading value	ReadInt/ReadPtr	Access variable, pass "raw" value through one of the AbstractState routines operatorReadFromIntVariableFilter/ operatorReadFromPtrVariableFilter
Writing value	WriteInt/WritePtr	Write value after passing through one of the AbstractState routines OperatorAssignmentToIntVariableFilter/ operatorAssignmentToPtrVariableFilter
Terminate program execution	TerminateExecution	No action.
Suspend and save	SuspendState	Copy current variables to a new set of variables.

state		

Table 29: The Parallel Implementation of Abstract State Primitives by AbstractState component and Variables.

Figure 53 shows a toy example that illustrates the application of control-flow related operations to both the *AbstractState* object and to each variable.

```
if (fMustProcesstateFallThrough_If_0)
    {
        tmpFallThrough = tmp.GLBWithTruePredicate(!tmpPred);
       tmp = tmp.GLBWithTruePredicate(tmpPred);
       stateFallThrough = statePrototype.CurrentStateCurrentInterface().DynamicSplit(tmpPred,1);
    }
```

Figure 53: Example of Abstract Semantics Code Exhibiting the Application of Control-Flow Related Operations to both the *AbstractState* object and to the Variables.

8.3 Software Modeling: Values

Program values carry their own state, exhibit characteristic behavior when acted upon by various operators, and are passed around and stored as units. Such features make a values an obvious candidate for modeling as an *object*. As discussed in Chapter 5, this thesis takes strong advantage of the capacity to represent values as objects, and in particular of the fact that all program values are accessed and manipulated through abstract interfaces. The approximation of run-time values by abstract values makes the translation process for value and expressions fairly straightforward: Values of type T in the program source are translated into values of a type corresponding to the abstract analogue of T. Thus values of type *integer* and *boolean* are mapped to values of type *AbstractIntValue*, and pointer and array values are translated into values of type *AbstractPtrValue*. The direct character of this mapping is carried over to the manipulation of values as well: As will be discussed further in Section 8.5, for the most part an expression within the standard semantics translates directly into a comparable expression in the abstract semantics.

8.4 Modeling Statements and Control Flow

Having briefly surveyed the encapsulation of the objects manipulated within a program, we turn now to the modeling of the computational constructs that manipulate such objects. This section examines the translation of expressions and statements into the abstract semantics. Before launching into these descriptions, it is worth pausing to understand the general means in which the abstract execution will proceed.

8.4.1 Preliminaries

8.4.1.1 Handling Uncertainty in Path of Execution

As noted in Appendix 1, the execution of a program within a particular execution context (set of external inputs) leads to a single execution trace through a program's source code. While this execution may execute code at a particular point in that source code any number of times, at any particular point in time there is always a unique locus of execution within the program.

By contrast, the abstract semantics is associated with the "execution" of a program under a *distribution* of possible execution contexts. Because the execution of the program under different contexts can lead to different paths being taken through the program, we can think of the abstract semantics execution as executing *all possible paths* through the program simultaneously. For example, while in the standard semantics a particular execution of a conditional will only evaluate *either* the consequent or the alternate side of the construct, within the abstract semantics *both* sides of a conditional may have to be evaluated. Similarly, a while loop that executes a particular number of times for a particular execution within the standard semantics may have to be modeled as executing an *arbitrary* number of times in the abstract semantics.

An essential constraint on the design of many of the translations of constructs to the abstract semantics is the need to correctly deal with non-termination of a path of execution. The fact that there are multiple possible paths of execution within the abstract semantics necessitates some care when modeling possible program termination and non-termination. Following the discussion of Chapter 2, we can model a path of execution whose execution never completes as returning $\perp_{state}$. The distinguished state $\perp_{state}$ is unique in that all constructs are strict when taking $\perp_{state}$ as an entry state (that is, for any statement s : *State*$\rightarrow$ *State*, s($\perp_{state}$) = $\perp_{state}$.) Therefore, if any construct s_i along a path $S = s_0 \circ s_1 \circ s_2 \ldots \circ s_n$ returns $\perp_{state}$, then S returns $\perp_{state}$. Within a standard semantics interpreter, such termination might be signaled by throwing a "bottom exception", which propagates up from its source to the top level of the interpreter and terminates the program.

As might be expected, the situation is more complex for an *abstract* semantics interpreter. In cases where we must deal with a number of different possible paths of execution, we must be able to accurately simulate the termination of a path and the propagation of a returned state $\perp_{state}$. Very importantly, the strategy must take into account the fact that the terminated path likely represents just one of many possible paths of execution. For example, consider the code for a conditional I: *State*$^{\#}\rightarrow$ *State*$^{\#}$ shown in Figure 54. I contains

two substatements, C and A, both with functionality $State^{\#} \rightarrow State^{\#}$. Suppose $C \equiv s_0 \circ s_1 \circ s_2 \ldots \circ s_n$, and during execution of C one of the s_i returns $\perp_{state}$. While this would be sufficient to terminate the execution of the program in a standard semantics execution of I, such an action would not be permissible here, where $I(s) = C(s) \sqcup A(s)$. Instead, when one of the s_i returns $\perp_{state}$ the system must terminate the execution of C, and propagate $\perp_{state}$ up to the level of I. I must "catch" the return of $\perp_{state}$ from C, and continue on to execute A, evaluate the least upper bound $\perp_{state} \sqcup A(s) = A(s)$ to find the value of I(s).

```
I(s) ≡
    if (T_boolean)
        C
    else
        A
```

Figure 54: Managing Non-Termination in a Conditional.

There are two techniques easily employed for the purpose of handling non-termination; and the decision as to which is desirable depends on the target language employed and the relative importance of code clarity and analysis efficiency. Each of these techniques is briefly examined here.

8.4.1.1.1 Exception-Based Handling

The first of the methods for handling path non-termination was hinted at by the comments above that likened the propagation of $\perp_{state}$ within the program to the propagation of a "bottom exception". The signal of such an exception causes execution to depart immediately from its standard path and to proceed outward through all constructs (if necessary, unwinding the stack) until an exception handler is reached. By placing exception handlers at each point where different paths of possible execution diverge, we can catch cases where one of those execution paths terminates and continue on to explore the other possible paths.

```
halt ⇒
            throw new BottomException
```

A statement that executed two substatements *A* and *B* independently and combined their results (e.g. by taking their least upper bound) could be translated as follows

```
f(A, B) : (State#×(State#→ State#)×(State#→ State#))→State# ⇒
Try {
    T[A]        // execute A to get resulting state
}
Catch (BottomException b)
    ;
StatePostA = CurrentStateSnapshot()  // could be bottom
Try {
    T[B]        // execute B to get resulting state
}
Catch (BottomException b)
    if (StatePostA.IsBottom)
        throw new BottomException
// process statePostA and current state (state after B)
```

Figure 55: Translation of Two Substatements and Subsequent Combination.

(Note that the underlined State# in the signature for *f* indicates that f is strict in its first (implicit) argument, namely the current state.)

Because exceptions propagate without difficulty across function boundaries and down the call stack, no special handling is required at the top-level of functions or surrounding function calls. Exception-based methods for handling non-termination have the virtue of simplicity of implementation and ready visual comprehensibility for one who is acquainted with the conceptual model of bottom-state propagation. Unfortunately, they share with all exception-based mechanisms a significant performance limitation. The performance penalty associated with exception handling is severe in C++ , and is significant in Java as well – the language of the sample implementation discussed here. We turn now to examine a higher-performance but somewhat more obscure methodology for handling exceptions.

8.4.1.1.2 A Higher-Performance Approach

While exception-based handling of control flow is convenient and easy to understand, it uses mechanisms that are gratuitously powerful for our needs; as discussed above, this power comes with high costs. Exceptions are designed to be used in a very general-purpose fashion. Most importantly, they are optimized to operate in contexts in which the identity of the surrounding exceptionally handler cannot be statically determined and where a variety of information can be returned via the exception. Our uses of exceptions above are much more modest: In *many* cases, the identity of the nearest enclosing exception handler can be statically determined. In those cases where it cannot, the enclosing handler can be easily discovered with a small amount of mechanism (see Section 8.4.2.1). In addition, the abstract semantics implementation has no need to propagate additional information from the point where the bottom exception is triggered to where it is caught. Given such conditions, the use of a more direct and lightweight mechanism seems more appropriate. Figure 56 shows how we can model path termination within code where the surrounding construct is statically known. Figure 57 demonstrates the general form used for a conditional construct in

such a system. (Note that the focus here is on the handling of the control flow, so the code necessary to evaluate and dispatch on the result of the value returned by the predicate has been omitted.)

```
halt ⇒  goto bottomHandlerLabel
```

Figure 56: Handling of a path-terminating construct within the more efficient system

```
if(p,A, B) : (State*×(State*→ State*)×(State*→ State*))→State* ⇒
    fTerminatedAᵢ = false
    …    // process predicate
    T[A]        // execute A to get resulting state
    StatePostA = statePrototype.CurrentStateCurrentInterface().SuspendState();
    goto ExecuteBᵢ

bottomHandlerForAᵢ:// any termination in A goes here.
    fTerminatedAᵢ = true
ExecuteBᵢ:
    T[B]        // execute B to get resulting state

    // process statePostA and current state (state after B)
    …
    goto nextStatementᵢ
bottomHandlerForBᵢ: // any termination in B goes here.
    if (fTerminatedAᵢ)
        goto bottomHandlerForThisConstruct
    else
        StatePostA.SetAsCurrentState(5);
nextStatementᵢ:
```

Figure 57: Example of a Construct Containing More Efficient Bottom State Handlers.

There may be points during the execution of a program at which the location of the nearest surrounding bottom state handler cannot be statically determined. For example, Section 8.4.2.1 describes how code size can be reduced when executing a conditional under the abstract semantics. For example, the code to execute the consequent of a conditional can be shared between the case where the predicate to the conditional is known to be true, and the case when the predicate's value is not known. When executing the code associated with the consequent it is not in general possible to know whether a return of bottom should terminate the entire execution of the construct (as would be the case if we are dealing with a known predicate), or whether a return of bottom should simply trigger the beginning of the evaluation of the alternative portion of the conditional (as would be the case when we are exploring a conditional with an unknown predicate). While the exact handler for a particular thread termination may not be statically determinable, a bit of additional work will let us exploit some pieces of knowledge very effectively. In particular, we know two things:

- The bottom handler is located within the same function. While this is not always the case *a priori*, we can guarantee this fact by ensuring that *every* function is associated with a "top level" handler for bottom states, and creating one if necessary.

- The bottom handler is directly nested in one of the statements enclosing the source of the bottom exception. Like the execution process itself, all bottom state handling occurs in a *hierarchical* fashion. If we cannot determine statically as to which bottom handler to go to, it is only because there are some handlers within the constructs containing the source for the bottom state are not active.

Given that the set of possible bottom state handlers is limited to those inside enclosing constructs within the function, it is possible to create our own specialized machinery for routing notifications that a bottom state has returned if necessary. Moreover, it is possible to do so using techniques simpler and (more importantly) cheaper than are employed in traditional, full-bodied exception handling. One obvious way of handling this is to have inactive bottom handlers simply "propagate up" any bottom state notification that reaches them. Section 8.4.2.1 provides an example of where such a technique is used. An alternative methodology is to manage our own stack of "bottom state handlers". The maximum depth of this stack can be determined statically, and placed into a set of local variable located on the run-time stack. The code for such a methodology is shown

```
switch (currentBottomHandler) {
    case topLevelBottomHandler:
        goto TopLevelFunctionBottomHandler1;
    case idLocalBottomHandler0:
        goto BottomHandler2;
    case idLocalBottomHandler1:
        goto TopLevelFunctionBottomHandler1;
...
}
```

Figure 58: A Path Termination Handler for Cases in which the Bottom State Handler is not Statically Known.

The only challenging case that remains is in the need to propagate a bottom exception interprocedurally. Fortunately, the need for interprocedural propagation is expected to small, for two reasons.

- We expect that evaluation in the abstract semantics will uncover many potential lines of execution, and that the large majority of terminated paths are likely to occur while other possible paths are awaiting execution (for example, inside of conditionals associated with predicates of unknown value or during the fixed point iteration of a loop or recursively defined function).

- As will be discussed in the next section, all path termination caused by control flow constructs is handled within the function in which it originates. The only cases that are likely to terminate an execution are those caused by explicit termination requests. While it is relatively common for control flow constructs to terminate paths of execution (see below), path termination by an error or an explicit statement or library call (e.g. *exit* in the standard C language library) occur far less frequently.

Given the low frequency of interprocedural path terminations, we can make special provisions for this exceptional case without severely affecting the execution time for the expected case. In particular, we can make limited use of the exception mechanism for propagation *between* procedures, while employing direct jumps when possible. Thus, the top-level bottom handler associated with a function could throw a *globalTerminatedPathException* exception in those cases where $\perp_{state}$ is to be returned from the function; the caller function would, in turn, have a *globalTerminatedPathException* exception handler wrapped around the call. An exception received from the call would transfer control to the appropriate bottom handler within the caller (using the higher-performance exception handling mechanism.)

8.4.1.2 Handling Unstructured Control Flow

One more set of important cases must be addressed –those associated with handling unstructured control flow. Examples of constructs giving rise to such unstructured flow include the *goto* statement in C and computed *goto*s in FORTRAN and BASIC. In general, the translations we use impose a top-down hierarchical flow ("recursive descent") on the execution of the program – in essence, an abstraction of the familiar "recursive descent" flow typically used in a standard semantics interpreter. Under this technique, construct *C* that begins executing proceeds to invoke its child constructs (e.g. substatements), which eventually return state to *C*. *C* may operate on the results of its child constructs (e.g. to take the least upper bound of their resulting states or to compare the result of evaluating an expression to zero), and then completes executing.

Although this technique is convenient to use and very well-suited to an abstract semantics in which multiple paths of execution are possible[43], it complicates handling of control flow that is not confined to hierarchical patterns of execution. For example, the handling of *goto* statements requires a two-step protocol and some additional bookkeeping machinery. In particular, in order to adhere to the recursive-descent model, a *goto* should not immediately transfer control out of the source construct to its target label. Instead, execution must continue and travel back up the source construct before it passes on to any target. (For example, there may be other pending paths of execution within the source construct.) In lieu of performing the transfer of control directly, the system must enqueue the target label to guarantee future execution, and return $\perp_{state}$ within the source construct.

[43] While it is possible to use other methodologies for executing constructs (such as the instruction-pointer-based methods illustrated in [Steensgarde, 1991 #107]), many abstract execution systems make use of hierarchical techniques similar to our own (for a recent and particularly impressive example, see [Chen 1994])

```
T[Goto X] ⇒
    stateLoopEnd.LUBToCurrentState(statePrototype.CurrentState());
      stateLoopEnd = statePrototype.CurrentStateCurrentInterface().SuspendState();
      throw globalTerminatedPathException;
```

Figure 59: Handling of Unstructured Control Flow Construct in an exception-based termination system.

Because $\perp_{state}$ is being returned, execution will then revert to the closest surrounding point at which the system is waiting to explore another thread. (As discussed in the previous section, this can be implemented within the abstract semantics code by means of an exception handler that catches the "bottom exception", or through some sort of *goto*-based mechanism.)

As execution continues, the target label that was reached earlier may again be reached in the course of running. In this event, the label is dequeued and need not be separately executed. If at some point all execution within the function has ceased, a top-level handler revives any pending constructs and executes them.

8.4.1.3 Preliminaries: Mapping Mechanics

The TACHYON compiler responsible for translating a program from the standard semantics to its analogue in the abstract semantics is engineered for easy modification.

8.4.1.3.1 Mapping Structure

In particular, the program operates by reading a set of user-specified code templates. Each of these templates specifies the abstract semantics analogue to a particular statement type. The system then uses these templates to perform the program translation. Whenever a statement is encountered, the appropriate template code is copied, parameterized, and emitted. The template parameterization process simply substitutes a set of instance-specific parameters into the template at points where appropriate markers have been left; such substitution can be undertaken for either statements or expressions. Within the current system, the parameterization is *ad hoc* in the sense that the routine for handling a particular type of statement is hard-coded to substitute certain types of information for pre-defined marker types.

Within the excepts from the mapping templates shown within this chapter, the markers indicating points of substitution are often indicated by pseudo-function calls, such as "STATEMENT_PARAMETER_1()" and "EXPRESSION_PARAMETER_1()". The appropriate parameter will be substituted for this entire expression when a particular instance of the construct to translate is encountered. The next section discusses another lower-level and more structured parameterization mechanism that is designed to help in the translation of statement operations.

8.4.1.3.2 State Mapping Macros

In addition to supporting parameterization markers, the templates used within TACHYON support a macro-like mechanism for emitting state-specific code. The motivation for these macros is the use of a "hybrid" state abstraction, as first described in Chapter 4. In particular, the representation of the abstract state is divided between an *AbstractState* object and the variables in the abstract semantics code that represent non-escaped variables in the user's source code. As described in Section 8.2, state operations are simultaneously applied to both the *AbstractState* object and to the variables. This poses a problem for a naïve implementation of the statement templates. In particular, while the template creator could code the desired state operations for the *AbstractState* object, there is no way to identify at template creation time the set of variables among which that state will be a distributed. The members of this set depend on the particular construct being considered.

The solution adopted by TACHYON is to allow the template creator to specify state operations at a higher level of abstraction. The parameterization machinery is then responsible for mapping the specified operation to both the state object and to the variables at the point of substitution (at which time all of the variables are known).

State Specific Macro Name	**Handling for AbstractState**	**Handling for Variables**
SplitCurrentState_PARAMETER_1	DynamicSplit	GLBWithTruePredicate method
SaveCurrentStateAndInstantiatePreexistingSavedState_PARAMETER_1	SuspendState, SetAsCurrentState	Copy current variables to a new set of variables. Set current copies of variables to saved-away copies of variables.
SaveGlobalCurrentStateAndInstantiatePreexistingGlobalSavedState_PARAMETER_1	Same as above	Same as above
LUBCurrentFromSavedState_PARAMETER_1	LUBToCurrentState	LUB method
LUBGlobalCurrentFromSavedState_PARAMETER_1	Same as above	Same as above
LUBSavedStateFromCurrentAndInstantiateSavedSt	FiniteHeightLUBFromCurrentStateA	FiniteHeightLUB method, assignment to current

ate_PARAMETER_1	ndSetAsCurrentSta te	variables from set associated with saved-away state.

Table 30: The State-Related Macros Used in TACHYON's Templates.

8.4.2 Modeling Statements

The section above described some of the challenges that arise during the translation of statements within a program to their abstract semantic analogues, and discusses the mechanisms by which these issues are handled. We now turn to take a more detailed look at how TACHYON handles the translation of different types of statements.

8.4.2.1 Conditionals

The first group of statement whose translation we will examine is the conditionals, which only two closely related types are supported – a conditional with both "then" and "else" branches, and one with only the "then" branch. The mechanisms used here are presaged by some of the discussions above, and anticipate the handling of the loop statement discussed in Section 8.4.2.2.

8.4.2.1.1 Single-Branched Conditional

The mapping for a conditional with only a "then" and no "else" branch is shown in Figure 60. The code executes the translation of the body of the conditional unless the predicate is known to be false (in which case no action is performed). The conditional body (expanded from the call to "STATEMENT_PARAMETER_1()") is thus executed in one of two conditions: If the predicate is known to be true, or if the predicate is neither known to be true nor known to be false. By using the same code for each of these circumstances, the code avoids the need to duplicate the translated "then" branch. The code instead makes use of a local variable – *fMustProcesstateFallThrough* – to dynamically maintain information on the context under which the "then" branch is being evaluated.

In the first case, execution of the body is direct; in the latter case, the abstract execution encounters a control flow split at the top of the conditional, and a join at the end of the statement. The *SplitCurrentState_PARAMETER_1* and *LUBCurrentFromSavedState_PARAMETER_1* macros apply the appropriate method calls to both allocated memory regions and variables. Note in particular that the split macro includes calls to the split methods of the *AbstractState* and applies the *GLBWithTruePredicate* operator to each of the variables used within the body. These calls allow the user's domains to sharpen their model of program state while executing in the loop. For the case in which the predicate is known to be true, no such epistemic "sharpening" is performed on the variables, since the predicate is already known to be true by the approximating domain.

Note that the code is required to include an exception handler around the statement body in order to handle any early termination during the execution of the abstract body translation (as would occur, for example, if the body contained a non-local transfer of control such as a "continue" or "break" statement to a loop surrounding the conditional, or a "return" statement).

```
int TranslatedIfNoElse()
    {
    class _IAbstractState stateFallThrough = null;
    class boolean fMustProcessstateFallThrough;

    DeclareState_PARAMETER_1(FallThrough, null);   // must replace parameter declarations
    tmpPred = EXPRESSION_PARAMETER_1();

    if (!tmpPred.FKnownFalse())        // if the predicate is false, just fall through
        {
        fMustProcessstateFallThrough = !tmpPred.FKnownTrue();

        // if we must also consider the possibility of falling through the construct,
        //  remember this beginning state
        if (fMustProcessstateFallThrough)

    // here, the continue case here is true if tmpPred == 1 at runtime; the current state takes
the other
            SplitCurrentState_PARAMETER_1(FallThrough, tmpPred, 1);

        // ok, now execute the body, being prepared in case it terminates
        try
            {
            STATEMENT_PARAMETER_1();
            // ok, we've gotten through the body.  if the predicate was imprecisely known,
            //  consider the possibility that we've just fallen through by LUBbing with the
            //  statement entry state

            if (fMustProcessstateFallThrough)
                LUBCurrentFromSavedState_PARAMETER_1(FallThrough);
            }
        catch (class TerminatedPathException bottomException)
            {
            if (fMustProcessstateFallThrough)

    SaveCurrentStateAndInstantiatePreexistingSavedState_PARAMETER_1(Nothing, FallThrough, LinePa
stStmtEnd_PARAMETER);
            else
                throw bottomException;
            }
        }
    }
```

Figure 60: The Translation for a Conditional with a Consequent but no Alternate Clause.

8.4.2.1.2 Double-Branched Conditional

Figure 61 shows the mapping template associated with conditionals bearing both a consequent and an alternative branch. As might be expected, the translated code is substantially larger and more complicated than for the single-branch case. This is reflects the fact that any *combination* of the two branches may have to be executed, and the requirement the code must accurately create the post-state regardless of what combination of internal branches was executed. This task is made more difficult by the need to avoid excessive space expansion due to duplicated translations of internal statements. In particular, to avoid a

code explosion, the translation of the conditional is restricted to only include a single translation of each of the consequent and alternative. As was the case for the single-branched conditional discussed in the previous section, in order to correctly join the paths of abstract execution at the end of the conditional the execution of the conditional is required to *dynamically* record which path of execution it followed. The code could be made more compact, but is spelled out in detail in order to allow for easier comprehension both by the modifier of the mappings and by the developer working with the abstract semantics code.

Broadly speaking, the translation first handles the "then" portion of the statement, and later the "else" branch. If execution terminates prematurely in either of these sections, additional dynamically maintained reasoning is required about how to proceed (thus the exception handlers surrounding the translations of the alternative and conditional sections).

Note again that the "split" required if both paths are possible – this split will initialize the variables and memory locations for each branch of the state. As was the case for the single-branched conditional, this split allows user domains to epistemically sharpen the domain's model during the paths of execution originating in either branch.

```
int TranslatedIfElse()
{
    class boolean fMustProcessY;
    class boolean fPendingX;
    class boolean fPendingY;

    DeclareState_PARAMETER_1(PreY, null);      // must replace parameter declarations
    DeclareState_PARAMETER_1(PostX, null);     // must replace parameter declarations

    fPendingX = false;
    fPendingY = false;

    tmpPred = EXPRESSION_PARAMETER_1();

    if (tmpPred.FKnownTrue())
    {
        fMustProcessX = true;
        fMustProcessY = false;
    }
    else if (tmpPred.FKnownFalse())
    {
        fMustProcessX = false;
        fMustProcessY = true;
    }
    else
        {
        fMustProcessY = true;
        fMustProcessX = true;
        }

    if (fMustProcessX)
        {
        if (fMustProcessY)
            // ok, split off a new path of execution
            // this is going to be the path relevant for the "false" path
            SplitCurrentState_PARAMETER_1(PreY, tmpPred, 1);
```

```
        try
            {
            STATEMENT_PARAMETER_1();
            fPendingX = true;
            }
        catch (class TerminatedPathException bottomException)
            {
            if (fMustProcessY)

        SaveCurrentStateAndInstantiatePreexistingSavedState_PARAMETER_1(Nothing,PreY, LineStmtSt
art_PARAMETER); //  setting current from the already-created Pre-Y state
            else
                throw bottomException;
            }
        }
    // from this point on, the code doesn't care if actually tried to execute
    //  X: executing X and having gotten bottom looks same as if didn't try X

    if (fMustProcessY)
        {
        if (fPendingX)

    SaveCurrentStateAndInstantiatePreexistingSavedState_PARAMETER_1(PostX, PreY, LineStmtStart_P
ARAMETER);      // this is directly saved away
        try
            {
            STATEMENT_PARAMETER_2();
            if (fPendingX)
                LUBCurrentFromSavedState_PARAMETER_1(PostX);
            }

        catch (class TerminatedPathException bottomException)
            {
            // if we have returned bottom, but did get a state back from X, use it
            if (fPendingX)
                // statePostX.SetAsCurrentState();

    SaveCurrentStateAndInstantiatePreexistingSavedState_PARAMETER_1(Nothing, PostX, LinePastStmt
End_PARAMETER);
            else
                throw bottomException; // if only executing Y, bottom is return value
            }
        }
}
```

Figure 61: Translation of the Conditional Statement into the Generic Abstract Semantics.

8.4.2.2 Looping

This section examines the handling of loop-related constructs: The while *statement* and the loop *break* and *continue* statements that interface closely with it. The handling of these statements builds strongly on the examples and principles discussed earlier in the chapter.

8.4.2.2.1 The Loop Statement

Of any of the statements examined thus far, the modeling of a loop construct raises the most challenges. The fundamental reason for the greater difficulty lies in the fact that while a loop can be infinite in duration, the system must provide the user with the option of guaranteeing analysis termination. This requirement imposes fundamental constraints on the design of the abstract domains (see Chapter 2). It is in program

loops – both iterative and recursive – that the fixed-pointing that forms the basis for abstract execution is reified in the translated code, and that benefit from the domain constraints.

In the TACHYON system, the situation is further complicated by the desire to provide the user with a full spectrum of choices over the granularity of approximation employed in the system, including the option of precisely simulating the program as it evolves in the concrete semantics. This requires the loop handling machinery to offer the capacity both to allow for unbounded execution and to prevent it, depending on the user's wishes.

The design of the code below reflects these issues. The code allows for successive execution of loop bodies without taking the least upper bound of loop entry or exit state until a user-specified number of iterations have taken place. Like the PARTICLE system [Osgood 1993], the abstract semantics allows the user to distinguish between counts of iterations made on the basis of loop predicates *known* to be true, and those made due to the fact that the loop predicate was not known to be either true or false. This allows for the user to dictate different thresholds at which to start fixed-pointing for loop iterations performed "speculatively" and those whose execution will definitively take place at run-time.

Once the specified threshold of iterations has been reached, the system begins a fixed-point iteration designed to soundly approximate the poststate resulting from *any* number of additional iterations of the loop. In order to guarantee that the fixed-point is reached within a finite number of iterations, the system must guarantee both that each abstract value is associated with only a finite height domain and that the total number of such values is bounded. TACHYON requires guarantees to this effect by domains implemented in the TACHYON run-time (See Chapter 2). In return, the analysis machinery must provide indication during those times in analysis at which the fixed-height constraints must be adhered to. The translated loop machinery provides this information through the calls to the least upper bound machinery (in methods calls to *FiniteHeightLUB* and *FiniteHeightLUBFromCurrentStateAndSetAsCurrentState* called within the expansion of the *LUBSavedStateFromCurrentAndInstantiateSavedState_PARAMETER_1* macro) and through calls to any allocation routines (among whose parameters is a flag indicating whether the surrounding loop is in the process of performing a fixed point).

The structure of the translated loop body below may be somewhat confusing. The first large section of the loop body is designed to test whether the loop predicate is false. If so, the code takes the least upper bound of the current abstract state with any accumulated loop end state and exits the loop.

In the event that we don't know that the predicate is false, the abstract state undergoes a split. To handle the case in which the predicate is false, the loop end state has its least upper bound taken with the split state (potentially "sharpened" by knowledge that the predicate is false). The other side of the split state continues on to execute the body of the loop (contained in the expansion of *STATEMENT_PARAMETER_1*()) and is then carried back to the top of the loop translation below. A final code suffix following the loop is designed to instantiate the saved-away loop end state as the statement poststate once the loop has terminated.

```
int TranslatedWhile()
{
//  class _IAbstractState stateLoopStart = statePrototype.BottomElt();
//  class _IAbstractState stateLoopEnd = statePrototype.BottomElt();
    int ctPerformedKnownIterations;
    int ctPerformedUnknownIterations;

    class boolean fPerformingFixedPoint = false;
    class boolean fRecordedPossibleLoopExit;

    ctPerformedKnownIterations = 0;
    ctPerformedUnknownIterations = 0;
    fRecordedPossibleLoopExit = false;

    DeclareState_PARAMETER_1(LoopStart, Bottom); // must replace parameter declarations
    DeclareState_PARAMETER_1(LoopEnd, Bottom);     // must replace parameter declarations
    DeclareState_PARAMETER_1(LoopContinue, Bottom);    // must replace parameter declarations

    while (true)
    {
        tmpPred = EXPRESSION_PARAMETER_1();
        //  point iteration (because

        if (tmpPred.FKnownTrue())
            ctPerformedKnownIterations = ctPerformedKnownIterations + 1;
        else if (tmpPred.FKnownFalse())// not relevant for cases where we are doing fixed pt
        {
            // ok, LUB the LoopEnd state to this one to accumulate it
            LUBCurrentFromSavedState_PARAMETER_1(LoopEnd);

    // ok, save the accumulated value away in the loop end state.  note that b/c we'll be instan
tiating
            //  that state immediately, we don't need to instantiate any other state.
            SaveCurrentStateAndInstantiatePreexistingSavedState_PARAMETER_1(LoopEnd, Nothing);

            fRecordedPossibleLoopExit = true;
            break;
        }
        else
        {
            // first, split the state according to the predicate

            // note that we can reach the state AFTER the loop with the state
            // ok, for now, we'll make the curent state the false predicate state
            SplitCurrentState_PARAMETER_1(LoopContinue, tmpPred, 0);
            // ok, now we LUB to the loop end state

            LUBCurrentFromSavedState_PARAMETER_1(LoopEnd);

            // ok, now save away the loop end state and instantiate the loop-
beginning state to continue

    SaveCurrentStateAndInstantiatePreexistingSavedState_PARAMETER_1(LoopEnd, LoopContinue, LineS
tmtStart_PARAMETER);

            // ok,  now handle this code
            ctPerformedUnknownIterations = ctPerformedUnknownIterations + 1;
```

```
            fRecordedPossibleLoopExit = true;


        // note that the current state is effectively split for the condition where tmpPred != 0
            }

            if ((ctPerformedKnownIterations >= GlobalSettings.Instance().CtMaxKnownIterations()) ||
                (ctPerformedUnknownIterations >= GlobalSettings.Instance().CtMaxUnknownIterations()))
            {
                // ok, if we're doing a fixed-point iteration, LUB with the initial state
                //  note that this also LUBs the conditional mask


        // Here, we're interested in knowing whether the result of the LUB was higher than the
                //  LoopStart state -- if we're no longer increasing
                //  the loop start state, then there's no need to continue.

                fPerformingFixedPoint = true;

                // LUB to the saved away state and set the loop start state to the LUBbed value

        LUBSavedStateFromCurrentAndInstantiateSavedState_PARAMETER_1(LoopStart, LineStmtStart_PARAME
TER, s_fChanged);

                // if we've converged, leave the loop and adopt the post-state approximation
                // includes
                if (!s_fChanged)
                    break;
            }

            try
                {

        // within this,  all "continues" are mapped to LUBs with the loop start state, breaks with L
UBs to the loop end state
                STATEMENT_PARAMETER_1();
                }
            catch (class TerminatedPathException bottomException)
                {

        // if we trigger a termination in executing the loop, this loop iteration will never continu
e.
                //  we need to exit the loop immediately
                break;
                }
        }   // matches while (1)

        // ok, set the loop exit state to any accumulated loop exit state
        //  (note that this is not necessary if we never accumulated loop exit state because
        //    there was never any chance that we'd exit
        if (fRecordedPossibleLoopExit)
        {

        SaveCurrentStateAndInstantiatePreexistingSavedState_PARAMETER_1(Nothing, LoopEnd, LinePastSt
mtEnd_PARAMETER);
        }
    }
```

Figure 62: Translation of the Loop Statement.

8.4.2.2.2 Continue

The *continue* statement is designed to work together with the structures maintained by the surrounding loop. The translation of the statement simply takes the least upper bound of the current state with the accumulated saved-away state that represents the loop start for the next loop iteration. This handling is the same regardless whether the loop is currently being fixed-pointed or approximated more precisely. The construct

then signals the termination of the path of execution, sending execution to the closest enclosing handler (typically, a surrounding construct with a split based on an unknown conditional). The translation of the *continue* statement is shown in Figure 63. Note that the terminating exception is wrapped in a pseudo-conditional to avoid any compiler complaints concerning the exception handling in the code (e.g. complaints that subsequent code could not be reached).

```
int TranslatedContinue()
{
    LUBCurrentFromSavedState_PARAMETER_1(LoopStart);
    SaveCurrentStateAndInstantiatePreexistingSavedState_PARAMETER_1(LoopStart, Nothing);

    if (true)
        throw globalTerminatedPathException;
}
```

Figure 63: Translation of the Continue Statement.

8.4.2.2.3 Break

The translation of the "break" statement is very similar to that of the "continue" statement, with the exception of the fact that the least upper bound is taken to the loop end state rather than to the loop beginning state. In addition, a boolean is set indicating that a loop exit request was encountered. As can be appreciated from Figure 64, the translation of the "break" construct has much in common with that of the "continue" statement shown in Figure 59.

```
int TranslatedBreak()
{
    LUBCurrentFromSavedState_PARAMETER_1(LoopEnd);
    SaveCurrentStateAndInstantiatePreexistingSavedState_PARAMETER_1(LoopEnd, Nothing);
    fRecordedPossibleLoopExit = true;

    // to prevent the compiler from issuing dead code errors  for following code
    if (true)
        throw globalTerminatedPathException;
}
```

Figure 64: Translation of the Break Construct.

8.4.3 Summary

This section has provided a brief overview of the handling of control flow constructs within the compiled abstract semantics code. The code emitted adheres to a familiar recursive-descent strategy for evaluating constructs that is common in interpreters, but differs from execution under the standard semantics in that many potential paths are possible through the program. Termination of a path of execution does not therefore in general lead to the termination of the entire program. Instead, the system must "catch" the propagation of the distinguished state $\perp_{state}$ that signals a non-terminating path, and continue on to explore other possible paths. We further explored two ways of implementing this conceptual framework: The current technique is based on exception handling and has the virtue of easy comprehensibility but can suffer

from performance shortcomings. By contrast, there is a strategy based on direct jumps (via *goto* statements) which is highly efficient, but is also somewhat confusing and is not supported by our currently favored target language, Java.

8.5 Modeling Value Operators

This section examines the means by which compilation translates operations on values into their abstract semantics analogues. While the translation of control flow constructs contains some subtleties, the mapping of most value operators is surprisingly straightforward. This section presents the general rules by which this mapping is accomplished, and examines the few cases in which translation is more difficult.

8.5.1 General Rules

8.5.2 Standard Operators

Table 31 and Table 32 show the translation rules used to map value operators into compiled code. Most operators can be translated directly into method calls, although the method call may take additional arguments beyond those specified to the original operator. (For example, calls to create routines provide as an additional argument the source location at which the allocation takes place.) The "Notes" columns in each table indicate those variables that require special handling; some of the more important such categories (e.g. variable accesses, allocation, etc.) are listed in the sections below. Figure 65 illustrates the translation of an expression to its abstract semantics analogue. In general, value operators are translated into method invocations on objects. The example shows this, as well as an exception: The translation of memory accesses (which involve method calls to the state. Note that while the constant was translated into a variable reference, no state call is required to access that variable.

```
a + b == 2
        ⇒
statePrototype.CurrentStateCurrentInterface().operatorReadFromIntVariableFilter(57,a,6).operator
Add(statePrototype.CurrentStateCurrentInterface().operatorReadFromIntVariableFilter(59,b,6)).ope
ratorEQ(tempId0);
```

Figure 65: An Expression and its Abstract Semantics Translation.

	Value Action	Method name	Notes
Integer Binary Operators	*	OperatorMul	
	/	OperatorDiv	
	+	OperatorAdd	
	-	OperatorSub	
	%	OperatorMod	
	&	OperatorArithAnd	
	|	OperatorArithOr	
	^	OperatorArithXor	
	>>	OperatorRightShift	
	<<	OperatorLeftShift	
Boolean Binary Operators	&&	OperatorLogicalAnd	Shortcircuiting not currently supported.
	||	OperatorLogicalOr	Shortcircuiting not currently supported.
Cross-type Binary Operators	==	OperatorEQ	
	!=	OperatorNE	
	<	OperatorLT	
	<=	OperatorLE	
	>	OperatorGT	
	>=	OperatorGE	
	=	Special Case (See Below)	
Unary Operators	~	OperatorArithNot	
	!	OperatorLogicalNot	
	-	OperatorNeg	
Miscellaneous	Variable Forward/Cross edge Join	LUB	Takes source location.
	Variable Backedge Join	FiniteHeightLUB	Takes boolean to indicate if changed and source location.
	Variable Split	GLBWithTruePredicate	Takes as argument the predicate which is guaranteed to be true for this path of execution. Also takes source location argument.
	Variable Identifier	Special Case (See Below)	Calls to filter routine
	Initialization w/constant	CreateFromConstant	Takes source location argument.
	Uninitialized declaration	CreateUninitialized	Takes source location argument.

Table 31: Mapping of Operators for Integers and Booleans

Category	Value Action	Method name	Notes
Pointer arithmetic	+ (Pointer offset by integer)	OperatorOffset	Argument is an abstract integer
	- (Pointer difference)	OperatorSub	Result is an abstract integer
Comparison	==	OperatorEQ	Result is an abstract boolean
	!=	OperatorNE	Result is an abstract boolean
	<	OperatorLT	Result is an abstract boolean
	=<	OperatorLE	Result is an abstract boolean
	>	OperatorGT	Result is an abstract boolean
	>=	OperatorGE	Result is an abstract boolean
Miscellaneous	Variable Forward/Cross edge Join	LUB	Also takes source location.
	Variable Backedge Join	FiniteHeightLUB	Takes boolean to indicate if changed and source location.
	Variable Split	GLBWithTruePredicate	Takes as argument the predicate which is guaranteed to be true for this path of execution. Also takes source location argument.
	Variable Identifier	Special Case (See Below)	Calls to filter routine
Creation	Call to allocation primitive	CreateFromAllocation	Takes as argument the source location and a memory region allocated by the analysis machinery. Note that the memory region allocation is performed with explicit knowledge as to whether a fixed-point is being pursued.
	& (Address of variable operator)	CreateFromAddressOfVariable	Passed the memory region and offset of the variable whose address is being taken by the analysis machinery. Also takes source location. Not yet fully supported by compiler.
	Uninitialized declaration	CreateUninitialized	Takes argument indicating source location.
Dereference	*	Special Case (See Below)	Mapping depends on whether specifying lvalue.
	[ ]	Mapped to * dereference	Array references and pointers are viewed as interchangeable.

Table 32: Operator Mappings for Pointer Values.

8.5.3 Non-LValue Variable Identifier References

The TACHYON compiler recognizes a variable reference as a variable read if it appears in the program text other than as the target for an assignment. If the variable is a non-escaped variable[44], the compiler should

[44] Due to time constraints, this condition is not currently checked by TACHYON, although it should be and would require rather little additional work to support.

translate this reference into a reference to the corresponding abstract semantics variable, surrounded by a call to a routine that filters the "raw" value read from the variable. (As discussed in Chapter 6, the intention of this filter call is to allow the user's domains control over the semantics of variable reads and writes.) Figure 66 below shows an additional example of a code fragment that illustrates such a translation.

```
var2
    ⇒
statePrototype.CurrentStateCurrentInterface().operatorReadFromIntVariableFilter(51,var2,6)
```

Figure 66: Translation of a Variable Identifier Reference

In the case of an escaped variable, the compiler should emit code that simply accesses through a specialized compiler-constructed pointer to that variable's value. Note that while convenient, this solution has the undesirable effect of using the same mechanisms that are employed the semantic rules for general memory access in order to read escaped variables. In particular, the call to *ReadInt* or *ReadPtr* will normally contain semantic rules related to memory reference, but is also used in the context of escaped variable reference. A refinement of this strategy might allow for a disentangling of these mechanisms. Due to time constraints, the TACHYON compiler does not currently emit correctly code for escaped variables, although adding this would require very little effort.

```
...
...&var2...
....
var2
    ⇒
...pVar2...
...
statePrototype.CurrentStateCurrentInterface().operatorReadFromIntVariableFilter(51,statePrototy
e.CurrentStateCurrentInterface().ReadInt(pVar2,6),6)
```

Figure 67: Translation of Reference to an Escaped Variable

8.5.4 Non-LValue Pointer Dereferences

Pointer deferences are handled by making a call to the appropriate type-specific "read memory" routine in the state interface (*ReadInt* for integer pointers and *ReadPtr* for pointers to pointers). Note that the source code location of the read is provided as an argument to the memory access method.

```
*p
    ⇒
stateProtoype.CurrentStateCurrentInterface().ReadInt(stateProtoype.CurrentStateCurrentInterfac
e().operatorReadFromPtrVariableFilter(49,p,6),6)
```

Figure 68: Handling of Indirect Pointer Reads.

8.5.5 Assignment

Within the language being abstractly interpreted, all assignment operators (=) occur at the topmost expression level (this is enforced through the application of a preprocessor to a more general language). The translation of the assignment operator into the abstract semantics depends on the type of left-hand side present. Because the preprocessor also maps array indexing into pointer offsets and indirection, there are only three possible left hand sides to consider:

- **A variable whose address is never taken.** In this case, the assignment is transformed into a simple assignment to the variable, but a call is made to the state interface's *operatorAssignmentToIntVariableFilter* or *operatorAssignmentToPtrVariableFilter* routine prior to the assignment. (As was originally discussed in Chapter 6, the goal here is to allow for semantic flexibility in defining the rules for variable writes by delegating control over the value to be written to an appropriate method of the state interface.) An example of such a mapping is shown in Figure 69.

```
var1 = var2;
    ⇒
var1 = statePrototype.CurrentStateCurrentInterface().operatorAssignmentToIntVariableFilter(49,va
r1,statePrototype.CurrentStateCurrentInterface().operatorReadFromIntVariableFilter(51,var2,6),6)
;
```

Figure 69 : Handling of Non-Escaped Variable on Left Hand Side of an Assignment

- **A pointer indirection.** Pointer dereferences can be mapped directly to the appropriate call to the state-interface method needed to write the element (*WriteInt* or *WritePtr*). Note that because the assignment is handled internal to the state, no "filtering" routine needs to be called. Figure 70 below illustrates the handling of an assignment with a pointer dereference on the right hand side.

```
*p = var2;
    ⇒
statePrototype.CurrentStateCurrentInterface().WriteInt(statePrototype.CurrentStateCurrentInterfa
ce().operatorReadFromPtrVariableFilter(49,p,6),statePrototype.CurrentStateCurrentInterface().ope
ratorReadFromIntVariableFilter(51,var2,6),6);
```

Figure 70: Handling of Pointer Indirection on the Left Hand Side of an Assignment.

- **A variable whose address is taken at some point.** Because escaped variables must be modeled as locations in memory and references to them as general pointers, this form of assignment should be handled analogously to assignment to a dereferenced pointer. As mentioned above, the TACHYON compiler does not yet support this, although extensions to do so would be straightforward.

8.5.6 Allocation

While the internal mechanics necessary to precisely simulate allocation can be involved, such details are hidden within the user's approximating state domain. As a result, translation of an allocation request is straightforward: A call is made to the appropriate type-specific allocation routine in the current state domain, passing an abstract integer specifying the size of the region to be allocated, a boolean indicating whether or not the surrounding loop is performing a fixed point, and the location of the allocation request in the source code. The fixed-point information is used to indicate conditions in which the allocation must adhere to a bounded-height abstract domain to ensure convergence in a finite time. The source code location is specified to allow for the initialization of the pointer value domains resulting from the allocation.

```
new int[10]
    ⇒
static _IAbstractIntValue tempId0 = GlobalSettings.Instance().PrototypeAbstractInt().CreateFromC
onstant(10,0);
...
stateProtype.CurrentStateCurrentInterface().AllocateIntArray(tempId0,fPerformingFixedPoint,6)
```

Figure 71: Handling array allocation.

8.6 Modeling Functions

This section briefly surveys the translation of functions in the standard semantics into functions in the abstract semantics.

8.6.1 Introduction

Given the general strategy of mapping objects in the standard semantics into similar objects in the abstract semantics (statements to statements, expressions to expressions, variables to variables), it should not be too surprising that functions in the user's source code are modeled by functions in the abstract semantics. Indeed, Chapter 2 established the conceptual basis for this modeling. The following sections examine some of the issues involved in the translation, including the conversion of the function body and of "return" statements. The general issues here are similar to those seen in the discussion of previous constructs, and the general techniques used to handle them are comparable. The final portion of this section discusses the manner in which TACHYON could model *recursive* functions, so as to guarantee convergence of the recursion within a finite time. This capability is not currently supported by the system, but would only require moderate effort to add in the manner indicated.

8.6.2 Function Wrappers

When translating a function, the TACHYON compiler emits a sequence of boilerplate code surrounding the translation of the code for the body of the function. This boilerplate code establishes bookkeeping variables and other infrastructure used by the code within the translated function body.

The boilerplate code also includes a top-level handler for terminated paths. This handler makes sure that any accumulated function exit states are properly returned from the function prior to termination. In the event that the execution of the code has terminated without encountering a return statement (as indicated by the function-wide boolean variable *fEncounteredReturn*), then a path termination exception is simply rethrown from the top-level handler. In the event that *fEncounteredReturn* is true, the system sets the accumulated function-end return state as the current state and returns. The return handler for non-void functions is further responsible for returning the accumulated abstract return value, as accumulated over the execution of the function body.

Figure 72 and Figure 73 below show the boilerplate used for void and non-void functions respectively.

```
int TranslatedVoidFunctionWrapper()
{
    class _IAbstractIntValue tmpPred;
    class boolean fMustProcessX;
    // here, we need to keep explicit track if we've reached a return stmt
    class boolean fEncounteredReturn = false;
    class boolean fPerformingFixedPoint = false;
    class _IMemoryRegion rTmpAlloc;
    class _IRegionContents contentsTmpAlloc;
    DeclareGlobalState_PARAMETER_1(FunctionEnd, null);   // must replace parameter declarations

    try
        STATEMENT_PARAMETER_1();
    catch (class TerminatedPathException bottomException)
        {
        if (!fEncounteredReturn)
            throw bottomException;
        else
            {

    SaveGlobalCurrentStateAndInstantiatePreexistingGlobalSavedState_PARAMETER_1(Nothing, Functio
nEnd, LineAtFunctionEnd_PARAMETER);
            }
        }
}
```

Figure 72: Boilerplate surrounding Translation of Void Functions.

```
int TranslatedNonVoidFunctionWrapper()
{
    class _IAbstractIntValue tmpPred;
    class boolean fMustProcessX;
    // here, we don't need a fEncounteredReturn, since we can tell from vResult if we reached a
return

    class _IAbstractIntValue vResult;
    class boolean fPerformingFixedPoint = false;
    // here, we need to keep explicit track if we've reached a return stmt
    class boolean fEncounteredReturn;
    DeclareGlobalState_PARAMETER_1(FunctionEnd, null);   // must replace parameter declarations
    fEncounteredReturn = false;

    vResult = (class _IAbstractIntValue) ((class _IAbstractValue) GlobalSettings.Instance().Prot
otypeAbstractInt()).BottomElt();

    try
        STATEMENT_PARAMETER_1();
    catch (class TerminatedPathException bottomException)
```

```
        {
        if (!fEncounteredReturn)
            throw bottomException;
        else
            {

    SaveGlobalCurrentStateAndInstantiatePreexistingGlobalSavedState_PARAMETER_1(Nothing, Functio
nEnd, LineAtFunctionEnd_PARAMETER);
            // stateFunctionEnd.SetAsCurrentState();

            return(vResult);
            }
        }
}
```

Figure 73: Boilerplate surrounding Translation of Non-Void Functions.

8.6.3 Function Call

Handling a non-recursive (and non-mutually-recursive) function call is straightforward: The translation of a function call with a set of parameters is simply a function call to the abstract analogue of the target function. The actual parameters to this function call are just the abstract analogues to the set of actual parameters in the standard semantics.

```
    bar(2)
    ⇒
    static _IAbstractIntValue tempId0 = GlobalSettings.Instance().PrototypeAbstractInt().CreateF
romConstant(2,0);
    ...
    bar(tempId0);
```

Figure 74: Example of Translation of Function Call

Calls to recursive or mutually recursive functions require more sophisticated translation. Particularly notable in this respect is the need for each such function calls to

- Accumulate statistics on the number of times that it has been called.
- Use the current return approximation in place of actually invoking the function.
- Take the least upper bound with the current state at time of call with the accumulated loop-entry state for that function.

The compiler does not currently support recursive function invocation, but Section 8.6.5 examines a simple strategy by which it could be handled.

8.6.4 Return Handler

The section above described the outermost wrapper of boilerplate that surrounds the translation of a function body. That wrapper made use of information put into place by return statements, such as the return state and

(for non-void functions) the return value. The communication between return statements and the function wrapper takes place through the function end state (shown in the translated code listings as a variable *stateFunctionEnd* of type *AbstractState*) and through the variable *vResult* representing the return value. The operation of the return statement reflects the fact that it may simply be one of many return statements encountered in the context of abstractly executing the current function. Because of uncertainties concerning the flow of execution in the program at runtime, abstract execution can continue after executing one return statement, and may encounter additional return statements during its processing of the remainder of the function. The return statement must therefore take the least upper bound of the state and value to be returned with any accumulated states and values that were returned by previously encountered return statements. At the termination of the function, the function wrapper will take up the accumulated values and return them.

```
void TranslatedNonVoidReturn()
{
    vResult = vResult.LUB(EXPRESSION_PARAMETER_1());
    LUBGlobalCurrentFromSavedState_PARAMETER_1(FunctionEnd);
    SaveGlobalCurrentStateAndInstantiatePreexistingGlobalSavedState_PARAMETER_1(FunctionEnd, Not
hing);
    fEncounteredReturn = true;
    throw globalTerminatedPathException;
}
```

Figure 75: The Return Handler for Non-Void Functions.

```
void TranslatedVoidReturn()
{
    LUBGlobalCurrentFromSavedState_PARAMETER_1(FunctionEnd);
    SaveGlobalCurrentStateAndInstantiatePreexistingGlobalSavedState_PARAMETER_1(FunctionEnd, Not
hing);

    fEncounteredReturn = true;
    throw globalTerminatedPathException;
}
```

Figure 76: The Return Handler for Void Functions.

8.6.5 Handling Recursion

An important challenge not handled in the code below is guaranteeing the convergence of recursive function calls. The issue for recursive loops are much the same as those for iterative loops, with the exception that recursive loops must potentially deal with not only with unbounded heap allocation but also unbounded *stack* allocation. Fortunately, this difference does not raise any serious additional complications in the approach to address the issue.

Section 8.4.2.2.1 noted TACHYON's philosophy of providing the user with a choice as to the level of precision with which loops are approximated. In particular, we seek to allow the user to specify levels of precision ranging from maximally accurate but potentially non-terminating analysis (analogous to the

techniques used by run-time analysis systems) to analysis in which fixed pointing begins immediately upon encountering a loop.

As a prerequisite to providing the option of guaranteed convergence of recursive loops in a finite time, the analysis must be capable of recognizing conditions in which loop fixed-pointing is required. In particular, it must maintain machinery to recognize when the code has performed an excessive number of recursive invocations. In the spirit of providing the user with fine-grained control over the analysis, the mechanism should distinguish certain recursive invocations from speculative invocations. (See Section 8.4.2.2.1) When the number of speculative or certain recursive invocations exceeds the respective user-specified thresholds, the analysis machinery will begin the fixed-pointing process, and set a flag indicating that fixed-pointing is taking place.

8.6.5.1 Fundamental Algorithm

Recall that iterative loops are conceptually can be implemented as tail-recursive recursive loops. As might be expected, fixed-pointing process for recursive loops involves the same basic patterns as that for iterative loops. But the handling of recursive loops contains some twists due to need to perform separate fixed-points to approximate not only the loop entry state but also the function return state. The analysis system's technique for approximating each of these states is described briefly below.

8.6.5.1.1 Function Entry State Approximation

During the calculation of the function entry state, successively general recursive calls are made until the function entry state ceases to change. More specifically, during this phase execution continues into each function call encountered. When such a call takes place, a least upper bound of the new loop entry state is taken with the accumulated loop entry approximation. If a change is detected, recursive invocations will continue. Note that because the loop entry state is becoming more general throughout this process, until the point at which convergence occurs, no complete return state approximation will need to be made.

8.6.5.1.2 Function Exit State Approximation

The second phase of approximation for the results of a recursive function call involves leveraging the already-existing function entry approximation in order to find the fixed-point of the *return* state of the call. Recall that a recursive function in general will depend on the return state of its own invocation; as a result, the return state approximation must be successively relaxed until a fixed point is reached. During this phase, the return state to the function is approximated in an increasingly general manner, and each successive approximation is used to derive a return state approximation by the surrounding invocation (i.e. by the activation that call it). This process continues until the return state approximation converges.

Phase two only executes once a stable approximation is achieved for function entry state. If during this process a recursive call is made whose entry state is not approximated by the previously converged entry state approximation, phase one resumes until a new, stable function entry state is reached. (In general, while we will refer to fixed-pointing of the entry state as "phase one" and that of the return state as "phase two", this is an oversimplification. In particular, calls exhibiting more than one recursive call site will typically exhibit interleaved derivations of the function entry and return states. Section 8.6.5.2.3 discusses an approach by which such multiple-entry recursive calls can be more precisely approximated).

8.6.5.2 Challenges

While the algorithm given above is straightforward, there are a number of subtleties that must be addressed. The next few sections describe these.

8.6.5.2.1 Taking the Least Upper Bound of States with Different Stack Sizes

8.6.5.2.1.1 THE DIFFICULTY

Within the first phase of recursive fixed-point iteration, the analysis machinery is responsible for taking the least upper bound of the function entry state at the start of successive activation records. While this is conceptually similar to the creation of the *loop* entry state approximation, it is complicated by the fact that within the stack is of different size within each of these states. Moreover, in order to guarantee convergence, we must be able to construct a finite-sized approximation to *any* possible activation record – a set of objects of potentially (theoretically) unbounded size.

8.6.5.2.1.2 THE SOLUTION

Fortunately, a starting point for a solution to this problem is to recognize that what is being sought here is easy to characterize functionally. In particular, we are looking for a model of the entry state stack for which the behavior of the analysis will approximate the behavior of the standard semantics execution for *any* possible entry state. We could legitimately create *any* model of the unbounded set of stacks, as long that model does not differ from any stack in ways that affect the outcome of the analysis. More operationally, the analysis machinery can legitimately make use of a very a simple stack model as long as the machinery behaves in a manner consistent with any (non-overflowing) stack.

We can combine this observation and the recognition that the portion of the stack most critical to analysis precision during a function call is the topmost (current) activation record to create a simplified but still sound model of the stack. In particular, we can make use of an abstract model of the stack in which only the topmost activation record is accurately modeled and can be manipulated by the code. Because all other portions of the stack will not be observed by the analysis code while the function is running, we can

maintain a very simple model of these regions without concern that it will affect analysis. For example, if a piece of code maintains pointers to variables in the deeper frames, those pointers will be generalized sufficiently that two pointers from successive stack frames show no difference. By "blinding" the code to the unbounded area of the stack, the area whose least upper bound must be taken will be of fixed size. More specifically, this region will be composed of the variables in the topmost stack frame, global variables, and the abstract model of the heap (itself guaranteed to be of finite size due to mechanisms discussed in Chapter 6).

8.6.5.2.1.3 APPLICATION OF THE SOLUTION

While low-level languages permit reliance on numerous aspects of the stack, it is straightforward to prevent these dependencies from affecting the course of the analysis described above. In particular, reflect that code running in an activation record within the language being modeled must reach any data either through a local or global variable. By taking the least upper bound of parameters and global variables within the entry state of successive activation records until convergence is reached, the code will be unable to rely upon details of stack structure and non-local stack data. Because parameters and global data have converged as well, successive invocations "look the same" to the code and are guaranteed to act identically. Because no non-local, non-global accesses exist following phase one of convergence, the calculation of the return state approximation does not have to concern itself with convergence of those references.

This section has briefly sketched how we can safely approximate any run-time stack size during analysis despite the presence of low-level mechanisms allowing for dependence on contents and size of the stack. We do this by creating successively general approximations to only the global variables and parameters passed to the topmost invocation. This characteristic allows us to simulate a theoretically unbounded collection of stacks of arbitrary size by a single analysis stack of finite size.

8.6.5.2.2 Dealing with Stack Bounds

The discussion above glossed over the fact that stack movement is required to perform the fixed-pointing that generates the approximations to both the function entry and function exit state requires. During the fixed-pointing of the entry state, successive activation records are pushed onto the stack. Activation records are successively popped off the stack during the fixed-pointing of the exit state. There are two potential problems with this approach.

- **Finite stack size limits.** In the case of abstract domains of collectively very tall height, the convergence process can be sufficiently long to cause stack overflow can during approximation. This is a matter of

some concern given the fact that analysis activation records tend to be considerably larger than their standard semantics counterparts due to the existence of additional modeling infrastructure.

- **Failure for Return State to Converge by Initial Activation Record.** Function entry state is approximated by means of invoking the routine with successively more general loop entry states. As long as the fixed-pointing of the function entry state does not lead to stack overflow, it is guaranteed to face no other limits, and recursive invocation can continue until it is converged. This is not so for the loop return state, which *returns* successively broad approximations until convergence. A difficulty here is that the loop-return state may not reach convergence by the point at which the call stack has returned to its original recursive activation record. (For example, the return type for the function may be associated with abstract domains of a much greater height than the arguments, and thus requiring a much larger time to converge.)

In order to avoid the latter difficulties, the strategy given above is modified to approximate the return state "in place". The algorithm accomplishes this by successively calling and recalling a recursively invoked function from the same activation record while the return state is still converging.

The idea is to conduct the return state approximation in a single activation record. The analysis machinery begins by saving away the function entry state and associating an initial $\perp$ return state approximation with the function called (or, rather, with the call site for that function call – see below). Rather than making the call when it is encountered, the return state approximation is used. Execution then continues until the current abstract function approximation would return. At that point, the current state is compared to the approximated state used instead of the recursive call. If a change is discovered, the execution of the current function begins execution again, using the new return approximation when the recursive call is encountered. Following the convergence of the return state, the new approximation is successively returned until the bottom call state.

Note that if stack size constraints are sufficiently binding, a similar "in-place" fixed-pointing methodology could be used to create the loop *entry* approximation as well. Although the thesis does not describe this approach, its formulation is very simple and similar to that used for the return state approximation. The one obvious disadvantage to such techniques is the need for additional looping superstructure to surround the function body in order to iterate the successive in-place approximations. This superstructure reflects the fact that we have converted an operationally *recursive* process into an *iterative* one.

8.6.5.2.3 Dealing with Multiple-Entry Recursive Calls

The methods discussed above can benefit from slight further modification to deal with the presence of distinct recursive calls within the same function body. In such situations, the function entry and return approximation derived for one call may not be legitimate for another. Although performance advantages could result from placing all such calls in the same equivalence class (as would result from following the guidelines sketched in preceding sections), it would be more accurate to allow for separate modeling of each call. Either option is available.

If all such calls are placed into the same equivalence class, it may be necessary to restart function entry convergence even after function return convergence has begun. This additional processing reflects the presence of a dependency between the function entry state of one call and the function return state of another. See Figure 77 for an example where such interleaved fixed-point iteration may be necessary.

By contrast, if function entry and return state approximations are stored independently each call site, then for any given call site it is guaranteed that phase one will terminate prior to the beginning of phase two.

```
int foo()
{
    tmp = foo(3);
    …
    foo(tmp)
    …
}
```

Figure 77: Multiple Call Sites May Require Restarting of Function Entry State Fixed Pointing.

8.6.5.3 Conclusions

This section has provided a whirlwind discussion of some of the issues that arises when modeling recursive functions. Although there are some challenges to soundly modeling recursive invocation, the sections above provide a safe technique that would require no extensions to the existing runtime. This approach could be implemented within the current system with only a moderate amount of effort needed for mapping and compiler modifications. This represents a desirable extension to the existing compilation system.

8.6.6 Conclusion

This section has presented the means by which the TACHYON compiler models functions in the abstract semantics program. Most issues in function modeling are straightforward, and the associated mappings are simple. The compiler currently does not currently handle the modeling of recursive functions, despite the fact that the machinery to do so is well understood. Only the compiler would require modification to support this approach. We now conclude the chapter with a discussion of the modeling of other constructs.

8.7 Variable Declarations and Initialization

The handling of the declarations of variables depends on whether the declarations include initialization. Declarations that include initialization from another expression are translated into abstract declarations with initialization from the translation of that expression. Figure 78 shows an example of such a translation. Variables that are declared without initialization are assigned explicitly created abstract values representing uninitialized quantities of the appropriate type. These abstract values are made up of abstract domains whose *CreateUninitialized* routines have been called. Figure 79 shows an example of an uninitialized declaration translated in this manner. Note that on the *CreateUnitialized* method must be called on some existing value which includes the appropriate value domains with which to create the new value. This is accomplished by invoking the routine on the prototype value for the appropriate type.

```
int var2 = 1;
    ⇒
static _IAbstractIntValue tempId0 = GlobalSettings.Instance().PrototypeAbstractInt().CreateFromC
onstant(1,0);
…
_IAbstractIntValue var2 = tempId0;
```

Figure 78: Translation of Declaration with Initialization

```
int var2;
    ⇒
_IAbstractIntValue var2 = intPrototype.CreateUninitialized(3);
```

Figure 79: Example of the Translation of a Declaration Lacking Initialization.

8.8 Global Variables

Global variables within the user's source code are translated into corresponding global variables in the abstract semantic program using the same mechanisms applied for local variables. This mapping retains the proper scoping of such values.

8.9 Module Prefix and Suffix

```
void TranslatedProgramPrefixWrapper()
{
    static class TerminatedPathException globalTerminatedPathException = new class TerminatedPat
hException();
    static class _IAbstractState statePrototype = GlobalSettings.Instance().PrototypeAbstractSta
teInstance();
    static class _IAbstractIntValue intPrototype = GlobalSettings.Instance().PrototypeAbstractIn
t();
    static class _IAbstractPtrValue ptrPrototype = GlobalSettings.Instance().PrototypeAbstractPt
r();
    static class boolean s_fChangedOut[] = new class boolean[1];
    static class boolean s_fChanged;
    static class _IAbstractState stateTmpSave;
}
```

Figure 80: The Program Prefix Template.

```
void TranslatedProgramSuffixWrapper()
{
}
```

Figure 81: Program Suffix Template.

Figure 80 and Figure 81 show the code templates that are emitted by the compiler for each module translated. Currently, TACHYON only supports single-module compilation, and thus the templates include both variables that are conceptually program-level and module-level. A multi-module version of TACHYON would logically be associated with a template for a *global* set of code to be included with a program as well, as another module. Currently, this is accomplished by maintaining pre-written code in another Java module (rather than as a template). By the time this code is invoked, the user has parameterized the *GlobalSettings* prototype information with the user's domains. The singleton prototypes *statePrototype*, *intPrototype* and *ptrPrototype* therefore contain the appropriate domains. The prefix code also initializes a few other analysis-wide variables, such as the path termination exception object (thrown every time that a path is terminated), and the boolean variable array (used as a form of "out" parameter when determining if convergence has been reached in a fixed pointing process).

8.10 The Model Compiler

The introduction described a second compiler that has been implemented as part of the TACHYON prototype. This compiler translates system dynamics models expressed in the declarative IThink™ model language into their abstract semantics analogues. Because the syntax of the source language is so simple, the mapping required is much simpler than for the case of a general-purpose programming language. The translation process therefore represents a simplification of that used by the compiler described in the first portion of the chapter. There are, however, some needs for support for additional types of constructs. This section provides a brief overview of the translation process and discusses the handling of some special cases.

8.10.1 The Language

Models within IThink™ are made up stocks (representing state variables), flows (representing derivatives to the state variables, or contributions thereof) and converters (representing intermediate calculations). Each flow and converter variable is associated with an explicit equation that describes how the current value of that variable is determined from the values of the quantities on which it depends. Stocks are associated with implicit equations governing their change from timestep to timestep, where the magnitude of the change is

dictated by the size of the input and output flows. During execution, each timestep updates every stock based on calculations made from the values of the stock variables in the previous timestep.

The language used to describe the models within the IThink™ framework is declarative, and expresses three general types of information:

- **Each stock as a function of the previous state.** An update rule is provided for stocks, specifying their current value in terms of their value in the previous state and the value of other model variables (all computed using information from the previous state)

- **The initial value associated with each stock.**

- **The equation specifying for all other variables (converter and flow variables).** All such computations are implicitly made based on the values of stocks within the previous timestep.

These equations describe relationships that must hold for each timestep during the execution of a model.

Expressions within the language are similar to those handled earlier in the chapter, but do include a few additional constructs. In particular, expressions may include:

- Table or ("graph") functions that map inputs according to an arbitrary, user-defined numeric mapping function.

- Conditionals expressed as "If then else" rules, similar to the Ternary operator "?:" in C or Visual Basic's "IIf" expression.

- A variety of time-parameterized functions that allow the user to associate a variable with a time varying output.

- A unary *INIT* operator that can be applied to stock names, and represents the *initial value* of that stock.

8.10.2 Translation

8.10.2.1 Fundamental Mappings

The translation of expressions and statements within the model compiler generally parallels that seen within the general programming language compiler discussed earlier in this chapter. However, the declarative character of the source language and its flexible ordering requires additional work to place the translated code into an explicit execution framework. This framework consists of an initial series of initializations,

followed by a loop that contains the per-timestep calculations and updates. Figure 82 depicts the general character of the translation template.

```
    All variable initializations
    for (abstractT = ValueConstant0_0; !(t.operatorGT(timeStop).FKnownTrue()); abstractT = abstr
actT.operatorAdd(abstractDT))
    {
        non-stock update statements
        stock update statements
    }
```

Figure 82: Simple Template used in Translating Model Code to the Abstract Semantics.

- All variables within the model are translated into local variables in the translated code. Stocks whose initial value must be preserved are associated with an additional variable designated to carry the original value of the stock.

- Stock initializations create initialization expressions to be associated with the declaration of the local variable associated with the stock. Such initialization expressions can depend on other initial stock values either directly or through intermediate variables.

- Stock update statements are translated as abstract assignments that call off to assignment methods of the abstract state, and adjust the current contents of the stock by some incremental amount (weighted by *AbstractDt*).

- Converter and flow equations are translated into both initializations and into per-timestep update rules. All such calculations compute based on the values of the stocks in the *previous* timestep.

One important difference between the translation of expressions in the model and general language compiler is that *reads* of variables within the model compiler are not compiled into corresponding "filter" calls to the abstract state. By contrast, *writes* to variables (including those associated with initializations) do receive analogous handling. The motivations for this discrepancy are purely pragmatic, and lie in the fact that while instrumentation of variable reads seemed plausible in the context of analyzing traditional programs, it is unlikely to be of much value when analyzing model behavior. Future versions of the model compiler may shift to include analogous instrumentation of variable reads, if it is proven desirable.

8.10.2.2 Ordering Issues

The equations included in a model description exhibit arbitrary ordering, and can include both forward and backward dependencies. By contrast, the code resulting from translating those equations into the abstract

semantics must respect the standard constraints imposed by execution ordering. In particular, the compiler to the abstract semantics must emit initialization expressions that depend only on prior initializations, and loop update statements (for stocks or intermediate variables) that depend only on stock values from the *previous* timestep.

In order to accomplish this, the compiler constructs graphs of variable dependencies, and topologically sorts them to determine the appropriate order of updates. Because stock equations may contain (textual, not semantic) circular dependencies, they require particular additional care. In order to allow one stock to be updated before the other without the other then making (illegal) use of the updated variable, it is necessary to "break" such circularity. The compiler accomplishes this by introducing new intermediate variables that preserve the earlier value of the stock beyond the point at which it is written. Currently no attempt is made to find the *optimum* point at which to break circular dependencies, a shortcoming that can lead to unnecessarily large numbers of intermediate variables in certain cases.

8.10.2.3 Non-Standard Variables

In the process of translation, the compiler introduces convenience variables for a number of purposes. The variables (some of which have already been mentioned) are listed in the following table:

	Variable Name	**Notes**
Individual	AbstractDt	Abstract Value containing the time step taken on each iteration.
	AbstractT	Abstract Value containing the current time.
	RgIds	Array of identifiers. Used for mapping variable names to UIds.
	RgUIds	Array of UIds corresponding to the identifiers in RgIds. Used in the same mapping process as RgIds.
	TimeStop	Time at which to stop simulation. Simulation ends when AbstractT.operatorGT(timeStop).FKnownTrue()
	StatePrototype	State object prototype, used for obtaining and dispatching off of the current state.
Variable Classes	ValueConstant*X*	Class of variables used to represent constants in the code. These variables are initialized once to avoid the cost associated with repeated initialization.
	*X*_InitValue	Variables preserving stock initial values.

	*X*_PreservedCurrentValue	Variables preserving the a stock's value in the *previous* iteration. Introduced to break circular textual dependencies between stocks.

Figure 83: Variables Used in the Model Compiler.

8.10.2.4 Generality

In a departure from the handling of loops detailed in Section 8.4.2.2, the mapping currently used by the abstract semantics compiler does not currently support fixed-point iteration of the time-step loop. While the use of an abstract semantics value for t and dt allow for approximation of the value of t, the current compilation strategy has no means of analyzing the effects of performing an *arbitrary* number of timesteps. While it is questionable whether results of much value would come out of such a generalization strategy in a numerical context, this situation can be easily remedied by making use of the loop templates described in Section 8.4.2.2.

8.11 Conclusions

While the compiled analysis described in this thesis avoids the interpretive overhead associated with determining program structure, it carries with it a less obvious sort of interpretive overhead: That arising from leaving dynamic the particular domains that are present during operation. This interpretive overhead is reflected not only in the dynamic binding of the method calls, but more subtly in the inability of the analysis algorithm to make use of a "worklist" algorithm such as that seen in the abstract interpretation work of [Chen 1994] and [Ayers 1993]. While software-engineering motivations may make a "compiled analysis" approach more desirable, it would be very desirable to understand the performance tradeoffs associated with this choice.

SECTION III: SAMPLE IMPLEMENTATIONS

Chapter 9 Example Collecting domains

9.1 Introduction

This chapter briefly surveys a number of sample abstract *collecting* domains that have been implemented as experiments to test the generality and usability of the domain interfaces. Most of the domains are quite compact and are designed more for testing than for industrial strength use. Nevertheless, their formulation was a big help both in demonstrating the generality of the domain interfaces and in indicating areas where the interfaces required refinement.

Despite the compact and simple character of most operations, the number of methods whose support is required is substantial. Space considerations therefore preclude full inclusion of the domain code within the text of this chapter. Thus, discussion of each domain focuses on a select and fairly uniform set of operations (The user is referred to the complete code for the domain in the appendix).

9.2 Expression History Domains

The first domain we consider here is a value-only domain that carries around with each value a record of the expressions used to compute that value. This domain is aimed more at helping users better understand code rather than for collecting information for use in later processing. More particularly, the domain is useful for re-expressing a sequence of code in a functional manner, in which each value is described as a functional combination of a basis set of values.

Because uncertain control flow can lead to a value having many possible derivations, the expression history domains are based on a more fundamental and general-purpose abstraction, the "set domain". In particular, each value is modeled either as the distinguished top value or as a set of possible strings, each of which corresponds to one or more particular derivation sequences. When a value consisting of n derivation sequences is combined with another value holding m sequences, the aggregate value is represented as a single unit rather than as mn multiple possible combinations of possible values.

9.2.1 Domain Element Creation

Currently, the "basis set" of values over which all values are described consists of external values and initialization constants. Figure 84 shows the code used by the domain for handling declarations that involve initialization by constants (*CreateFromConstant*) or no initialization (*CreateUninitialized*).

```
public  _IIntDomainInfo CreateFromConstant(int v, int srcLoc)
{
    String str = "" + v;
    return(new ExprHistoryIntDomain(false, str));
}
public  _IIntDomainInfo CreateUninitialized(int srcLoc)
{
    return(new ExprHistoryIntDomain(false, "#"));
}
```

Figure 84: Code to Initialize the Expression History Domains.

In order to keep the height of the domains finite, the system assigns a maximum set size for each value; any value possessing more than that many possible derivations are abbreviated with the catchall top element.

9.2.2 Modeling Value Operators

Value combination via operators is straightforward: The aggregate expression is built up from each of the components in the desired manner (prefix, infix or postfix for binary values). Particular binary operators simply call off to a general routine *LocalBinaryCombine* that actually generates the appropriate string.

```
    public _IIntDomainInfo operatorAdd(_IIntDomainInfo argRHS) { return(this.LocalBinaryCombine(
(ExprHistoryIntDomain) argRHS, "+").operatorAddCustomFilter()); }
    public _IIntDomainInfo operatorLogicalNot() { return(LocalUnaryProcess("!")); }
    protected  ExprHistoryIntDomain LocalBinaryCombine(ExprHistoryIntDomain arg2, String strOp)
    {

    return(new ExprHistoryIntDomain(this.FIsTopElt() || arg2.FIsTopElt(), this.Unparse("", "|","
") + " " + strOp + " " + arg2.Unparse("", "|","")));
    }
    protected  ExprHistoryIntDomain LocalUnaryProcess(String strOp)
    {

    return(new ExprHistoryIntDomain(this.FIsTopElt(), strOp + " " + this.Unparse("", "|", "")));
    }
```

Figure 85: Example Value Combination Routines associated with the Expression History Domain and Their Supporting Infrastructure

9.2.3 Handling the Least Upper Bound Operator

Taking the least upper bound of two values is also very simple: If both operand values are non-top values with m and n possible (top-level) derivations, the result is a value with $m+n$ derivations. An exception to this rule applies if the resulting value would be too large – in this case, result value is approximated as the top value. The code to perform the least upper bound is shown in Figure 86. Note that the test against the maximum set size enforces a maximum height on the domain needed to guarantee convergence of fixed-point operators in program loops.

```
public _IValueDomainInfo FiniteHeightLUB(_IValueDomainInfo arg2, boolean fChangedOut[])
{
    SetDomain arg2Narrowed = (SetDomain) arg2;

    if (this.FIsTopElt())
    {
        fChangedOut[0] = false;     // this is top, so cannot have LUB(this, current) > this
        return(this);
    }
    else if (arg2.FIsTopElt())
    {
        fChangedOut[0] = true;      // this is not top, so must  have LUB(this, top) > this
        return(arg2);
    }
    else
    {
        Vector vNew  = MergeVectors(this.m_rgObjects,arg2Narrowed.m_rgObjects, fChangedOut);

        if (vNew.size() < TopSetSize())
            return FactoryMethod(false, vNew);
        else
            return FactoryMethod(true, null);
    }
}
```

Figure 86: Code to take the Least Upper Bound of Two Elements in the Expression History Domain.

9.2.4 Conclusion

While the expression history domain is a toy domain designed to probe the expressiveness of the value domain interfaces, it can offer practical help to numerical programmers. Indeed, it automates and simplifies an *ad hoc* technique that LISP programmers have used for years to help debug numeric code. [Sussman 1999] The simplicity of the domain demonstrates the very high leverage afforded by using an appropriate set of domain interfaces – given the shared "set domain" infrastructure used by this and several other domains, implementation of the custom rules for this domain takes very little time indeed.

9.3 Initialization Status Domains

9.3.1 Introduction

In this section, we examine the implementation of another value domain, this one designed to collect information on the initialization status of values being manipulated, thereby allowing the user to spot potentially risky reliance on uninitialized values.

As for the expression history domain, the quantum of information storage here is the *value* rather than lvalues (e.g. for variables). This allows for a flow sensitive analysis that is less likely to issue spurious warnings than a technique that always conflates different executions of the same code. Because the reasoning is all value-based and because warnings are simply issued dynamically rather than being

accumulated in locations, no significant state component is needed to implement the required domain semantics.[45]

9.3.2 *Abstract Domain Structure*

The three-level domain employed when judging initialization status is shown in Figure 87. Each of the points on this "diamond-shaped" domain classifies a given value into one of four categories. The leftmost position on the diamond corresponds to a classification of a value as having been computed from paths relying entirely on initialized values; an element holding the rightmost position signifies that the value has been computed from paths depending entirely on *un*initialized values. A value that is derived from paths dependent upon both initialized or uninitialized values will be associated with an element located at the top of the lattice. The bottom value indicates a value that is neither initialized nor uninitialized.

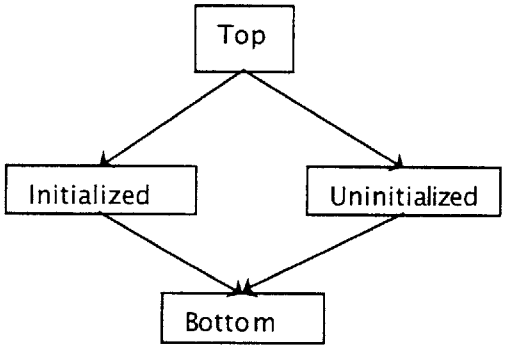

Figure 87: The Structure of the Abstract Domains for the Initialization Status Domain.

9.3.3 *Implementation Subclassing*

Just as the expression history domain relied upon a more basic and general "set domain" abstraction, the implementation of the initialization status domain is based directly on a more general "fuzzy boolean domain", which implements the isomorphic lattice shown in Figure 88. One piece of the wrapping shell is shown in Figure 89.

[45] Note that by avoiding the use of a state interface, the domain currently fails to recognize cases where *control flow* is shaped by an uninitialized or possibly uninitialized value. Extending the analysis ttake such cases into account would be straightforward, but

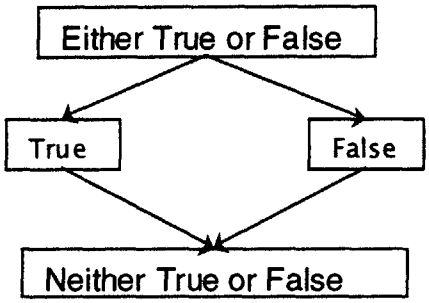

Figure 88: The Structure of the Abstract Domains for the Fuzzy Boolean Domain.

```
public static String UnparseFromClassification(int classification)
  {
      switch (classification)
      {
      case DefinitelyFalse:
          return("initialized");
      case DefinitelyTrue:
          return("uninitialized");
      case EitherTrueOrFalse:
          return("possibly uninitialized");
      case NeitherTrueOrFalse:
          return("illegal value");
      default:

      GlobalUtility.Assert(false, "Unrecognized FuzzyBooleanPtrDomain classification value!");
          return("illegal value");
      }
  }
```

Figure 89: Textually Rendering an Element of the Initialization Status Domain

9.3.4 Operation Implementation

The simplicity of the value domain structure of the initialization status domain is mirrored by simplicity of the implementations of the value domain operations. In order to allow sharing and reuse by other similar classes, many of the operations shown below are actually implemented in the Fuzzy Boolean Domain superclass.

9.3.4.1 Domain Element Creation

Figure 90 shows the routines responsible for creating elements of the initialization status domain in response to a value creation request. A request to create a constant value during analysis yields an initialized value

would erquire the addition of a state domain.

(as signified by passing "false" to the *FuzzyBooleanDomain* constructor). A request to create a value lacking initialization entails the creation of a domain element representing the uninitialized state (in which case "true" is passed to the constructor). These two methods illustrate how the domain handles the "base case" of value manipulation.

```
public _IIntDomainInfo CreateFromConstant(int v, int srcLoc)
{
    return new InitStatusIntDomain(false); // definitely NOT uninitialized
}
public _IIntDomainInfo CreateUninitialized(int srcLoc)
{
    return  new InitStatusIntDomain(true);
}
```

Figure 90: Routines to Create Domain Elements for Initialized (⊤) and Uninitialized (⊥) Values.

9.3.4.2 Modeling Value Operators

The handling of value combination requests is only slightly more complicated than the handling of value creation. Figure 91 shows how the code for this domains approximates the result of an adding two integer values. The code to implement the operators is operator-invariant -- the handling of all binary operator is uniform, making no distinction between the particular operations being emulated. In particular, the *operatorAdd* method simply calls off to the *BinaryCombine* method used to handle all binary value combinations. The *BinaryCombine* method effectively takes the least upper bound of the two elements in the domain lattice that was shown in Figure 88. The rules are as follows: If a bottom value is combined with another value, the other value is the result. If a definitively uninitialized or definitively initialized value is combined with a like value, the result is of the same type. Otherwise, the resulting value is either uninitialized or initialized.

Note that the calls to the *FactoryMethod* routine in Figure 91 are present only to ensure that an instance of the appropriate subclass of Fuzzy Boolean Domain is created. These methods are implemented as abstract methods that must be implemented by any class that derives from the Fuzzy Boolean Domain abstract class.

```
public urtinterfaces._IIntDomainInfo operatorAdd(urtinterfaces._IIntDomainInfo argRHS)
    { return(this.BinaryCombine(argRHS)); }
protected  _IValueDomainInfo BinaryCombine(_IValueDomainInfo arg2)
    {
      FuzzyBooleanIntDomain argRHSNarrowed = (FuzzyBooleanIntDomain) arg2;

      int classArg1 = this.Classification();
      int classArg2 = argRHSNarrowed.Classification();

      if ((classArg1 == NeitherTrueOrFalse) || (classArg2 == NeitherTrueOrFalse))
          return(FactoryMethod(NeitherTrueOrFalse));
      else if ((classArg1 == EitherTrueOrFalse) || (classArg2 == EitherTrueOrFalse))
          return(FactoryMethod(EitherTrueOrFalse));
      else if (classArg1 == classArg2)
            // if both are either DefinitelyFalse or DefinitelyTrue,
            //that's the class of the result
          return(FactoryMethod(classArg1));
      else
          return(FactoryMethod(EitherTrueOrFalse));
    }
```

Figure 91: The Implementation of Value Combination Routines in the Fuzzy Boolean Domain.

9.3.4.3 Handling the Least Upper Bound Operator

Figure 92 shows the routine used to implement the least upper bound operator in the Fuzzy Boolean Domain class. In order to allow for ascertaining whether a fixed-point has been reached, this method takes on an extra responsibility. In particular, the method not only takes the least upper bound of two values in the lattice shown in Figure 88, but also indicates whether the resulting value is higher in the lattice than the left hand argument ("this" in the routine).[46] Note that in order to reuse code the *LUB* method makes use of this same routine, despite the fact that it does not require the full generality of returning the boolean value.

```
public IValueDomainInfo FiniteHeightLUB(_IValueDomainInfo argRHS, boolean[] fChangedOut)
  {
      FuzzyBooleanDomain argRHSNarrowed = (FuzzyBooleanDomain) argRHS;
      int classArg1 = this.Classification();
      int classArg2 = argRHSNarrowed.Classification();

      if (classArg1 == NeitherTrueOrFalse)
      {
          fChangedOut[0] = !(classArg2 == classArg1);
          return(FactoryMethod(classArg2));
      }
      else if (classArg2 == NeitherTrueOrFalse)
      {
          fChangedOut[0] = false;
    // we know that classArg1 must be equal or higher in the lattice
          return(FactoryMethod(classArg1));
      }
      else if (classArg1 == EitherTrueOrFalse)
            // ok, here arg1 is at the top of the lattice
      {
            // can't get any higher than arg1
          fChangedOut[0] = false;
          return(FactoryMethod(EitherTrueOrFalse));  // nothing has changed
      }
```

[46] Note that in a pattern seen commonly within the implementation of the TACHYON runtime, the routine takes a single-element array as an argument. Java lacks a direct means of achieving pass-by-reference semantics for variables, and this techniques serves as a somewhat awkward substitute.

```
    else if (classArg2 == EitherTrueOrFalse)
      // ok, here arg2 is at the top of the lattice, and arg1 is NOT
    {
          // if we got here, we know that arg1 is NOT top
        fChangedOut[0] = true;
          // nothing has changed
        return(FactoryMethod(EitherTrueOrFalse));
    }
    // ok, if we're here, we know that both arg1 and arg2 are neither top nor bottom
    else if (classArg1 == classArg2)
    {
              // ok, if they are both the same classification,
              // that's the class of the result too
        fChangedOut[0] = false;
              // the reuslt is the same as arg1
              // nothing has changed
        return(FactoryMethod(classArg1));
    }
    else if (classArg1 == classArg2)
    {
        // ok, if they are both the same classification,
        //that's the class of the result too
        fChangedOut[0] = false;
          // the reuslt is the same as arg1
          // nothing has changed
        return(FactoryMethod(classArg1));
    }
    else
    {
        fChangedOut[0] = true;       // we know that arg1 was either true or false
        return(FactoryMethod(EitherTrueOrFalse)); // nothing has changed
    }
}
```

Figure 92: The Routine Taking the Least Upper Bound of Two Values in the Fuzzy Boolean Domain.

9.3.5 Conclusion

This section has examined another collecting value domain that can be implemented in a very simple manner. Customization of the value domain proves a very convenient mechanism for obtaining the desired semantics in the context of a high-fidelity analysis, and subclassing an existing implementation (that for the Fuzzy Boolean Domain) allowed for a particularly simple implementation. Appendix 3 provides a pointer to the complete code associated with this domain. As can be readily appreciated from that code, the presence of appropriate interfaces can allow for simple and expressive customizations with only a modest amount of work.

9.4 Code Generation Domains

9.4.1 Introduction

The most sophisticated abstract collecting domain yet implemented is a domain that generates code in the process of performing abstract execution. Within this domain, the abstract value domain elements are basic blocks of code generated for the expression that created the value. The abstract state domain elements are control flow graphs in which each node is a basic block, and each edge represents an explicit or implicit

transfer of control. The abstract state domain element representing the fixed point of the program execution contains the code to be generated for the entire program.

Because this domain is more complex than those presented above, the design has a larger number of important pieces. To allow the presentation to communicate some understanding of the domain's operation, more components of the domain will be presented. The presentation will survey the functionality associated with both value and state domain elements.

9.4.2 Abstract Domains

Unlike many of the domains presented above, this domain heavily customizes both the abstract value and the abstract state domains. Each of these is associated with a distinct abstract domain structure. It is worth recalling from Chapter 2 that collecting state domains represent abstractions of *sequences* of abstract states through which the program could have passed, and collecting value domains correspond to abstractions of the value calculation process. The design of the abstract code generation domains strongly reflects this modeling of state and value history.

9.4.2.1 Value Domains

As mentioned above, the abstract domain structure for the value domain consists of a basic block of code required to calculate the value, and an indication of the pseudo-register[47] in which the result of that calculation is to be found. Figure 93 shows the lattice associated with the value domain. In essence, a domain element containing a code sequence s can be interpreted as an approximation to all code sequences beginning with s. Thus, the lattice notion of "approximation" in use here is closely related to prefixing: We write that $a \sqsubset b$ iff b is a prefix of a.

As is in any value domain, combining value domains using value operators (e.g. "+") yields new value domains; here, the appropriate code sequences are simply "stacked" and the registers holding the current values combined with the appropriate assembly language instruction. There is one particularly important class of rules for value manipulation: Those relating to storage access. In particular, when values as *written* to a state, the registers used to hold their value are recycled, the code used to calculate the value is placed into the current basic block of the current state (see below). During storage, the code generation value domain elements are associated with the $\perp$ element. Thus, a value *read* from a variable is associated with only the code necessary to perform the read.

[47] For simplicity, all code manipulates pseudo-registers that can be assigned to real registers on the fly. See [Engler 1996] for an indication as to how such a mapping can be done efficiently.

Note that because the current system only takes the least upper bounds of values that are currently the *contents of locations* (variables or other memory contents), at no point is there a need to combine two values holding code sequences. Thus, the abstract domain structure of Figure 93 is more of conceptual than operational interest: There is no need to use the structure to guide lattice-related domain element combination.

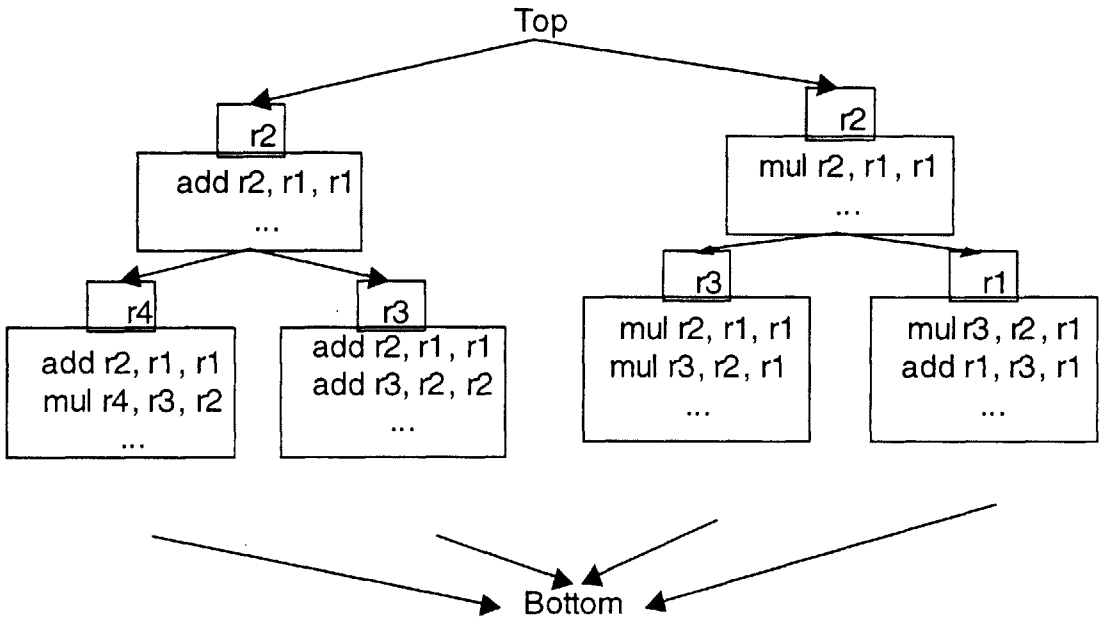

Figure 93: The Abstract Value Domain Member of the Code Generation Domains.

9.4.2.2 State Domains

Figure 94 schematically depicts the abstract state domain structure for the code generation domain. Abstract state histories are modeled as a control flow graph of basic blocks. As might be expected, control flow splits during abstract execution are lead to the presence of nodes with more than one child; joins are establish nodes with more than one parent.

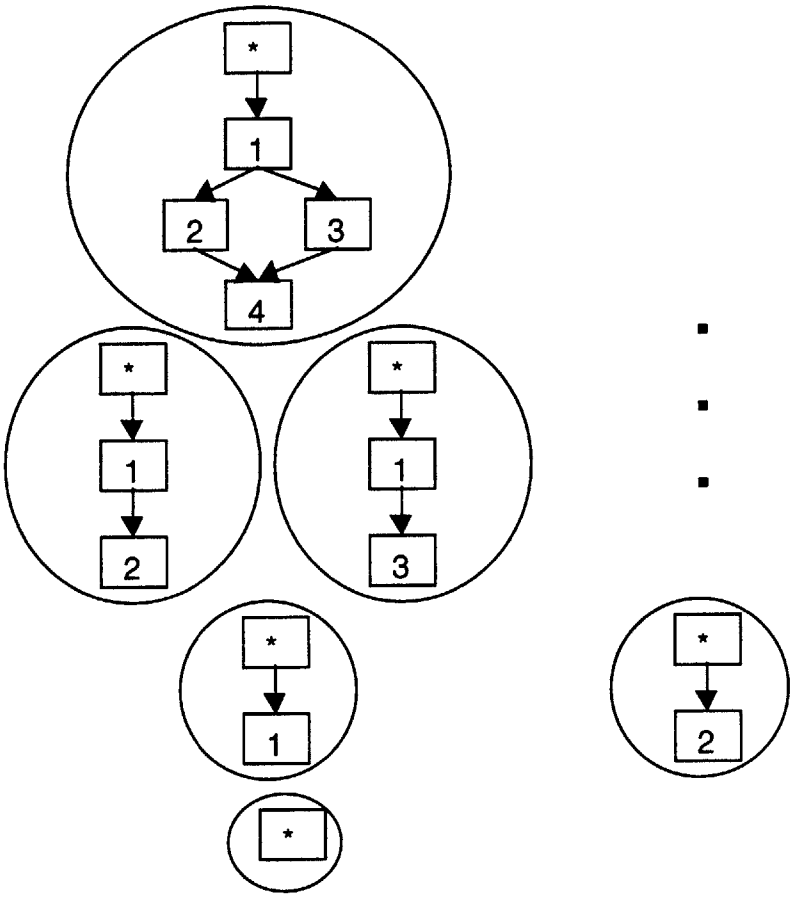

Figure 94: The Structure of the Abstract State Domain for the Code Generation Domain.

The notion of approximation in this lattice is closely related to graph containment. The result of taking the least upper bound of two domain elements (graphs) is a graph that soundly approximates (contains) either of the operand graphs. Note that in general, the graphs whose least upper bound is being taken will share some nodes and edges (representing common points of execution in the execution history) and will not share others. An example of the application of the least upper bound operator is shown in Figure 95.

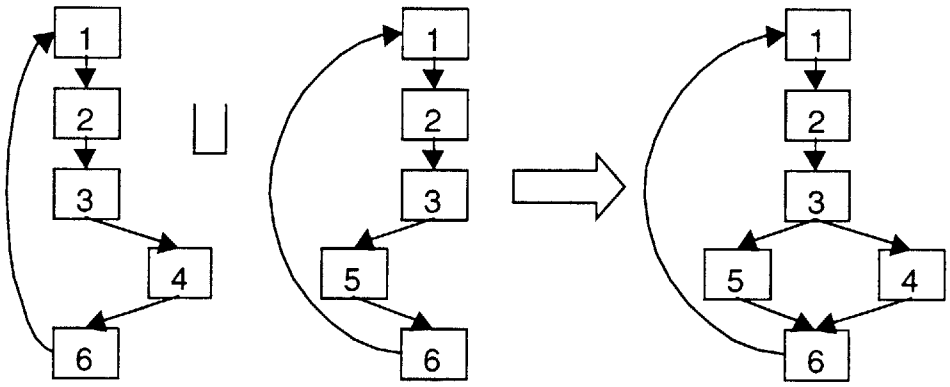

Figure 95: Application of the Least Upper Bound Operator to Elements of the Code Generation State Domain.

9.4.2.3 Comments

It is important to realize that while the control flow graph and basic blocks accumulated by the code generation domain do maintain a close conceptual relationship with the control flow graph and basic blocks of the program being analyzed, they can differ strongly in their realization. As discussed below, the control flow graph and basic blocks collected by this domain represent *specializations* of those associated with the original program. While the relationship between the full control-flow graph for the program being analyzed and that of the residual program is conceptually simple, the latter is not simply a subgraph of the former. In particular, the nodes of the specialized call graph may differ substantially from than those of the general call graph due to the shifting of some calculations and control flow to analysis-time, and the eliding of some control flow directives known not to be taken at run-time. Section 9.4.5 below discusses this domain from the perspective of program specialization in more detail, and ties it in with some recent contributions towards performing efficient run-time code generation.

9.4.3 Value Operation Implementation

This section examines the operations associated with the value domain. Section 9.4.2.1 noted that until the point at which they are written to a location (variable or other memory location), value elements keep track of two pieces of information. In particular, they maintain a basic block of code expressing the calculation that gives rise to them, and a register holding the result of that calculation. This section examines the

manner in which these domain elements are built up, and demonstrates how the execution of the domain results in the specialization noted above.

9.4.3.1 Domain Element Creation

Figure 96 shows the code that creates a new value domain element from a manifest constant and from a declaration lacking an initializer. Both of these types of declarations are translated into a "load immediate" instruction to a newly reserved pseudo-register. Because this routine is implemented in a type-generic value superclass, a call to a factory method *Factory* is used to create the member of the appropriate concrete class to carry the new value element that results. Note that for the sake of generating code, uninitialized values are simply modeled as initializations to zero[48]. A less deterministic result could be obtained by simply reserving a register without using any code to initialize it.

```
    public _IIntDomainInfo CreateFromConstant(int v, int srcLoc)
    {
        return((_IIntDomainInfo) CreateImmediateLoadValueOfSameType(v));
    }
    public _IIntDomainInfo CreateUninitialized(int srcLoc) { return((_IIntDomainInfo) CreateImme
diateLoadValueOfSameType(0)); }
    protected SCGValue CreateImmediateLoadValueOfSameType(int v)
    {
        Register rOut = Registers.AllocateNewRegister();
        OpcodeSequence vectOpcodesNew = new OpcodeSequence();
        vectOpcodesNew.Append(new LoadImmediateRTLInst(v, rOut));
        return this.Factory(rOut, vectOpcodesNew);
    }
```

Figure 96: Code to Create a Member of the Code Generation Value Domain From a Declaration with a Manifest Constant or No Initializer.

9.4.3.2 Modeling Value Operators

The code to approximate the results of a value combination is shown in two pieces in Figure 97 and Figure 98. The most notable characteristic of the code shown in Figure 97 is located in the routine *LocalBinaryCombineOrShortcut*, where the code checks to see if the current operation is known to yield a concrete result in the approximating domain.

If the approximating domain is indeed returning a concrete result, there is no need to emit residual code to actually *calculate* the result of the operation at runtime. In this case, the calculation has already been performed by the approximating domain during this analysis, and the system can simply generate code to load the value computed by the approximating domain. The code for this case is generated through the call to the method *CreateImmediateLoadValueOfSameType*, which returns an element of this domain containing

[48] Note that the situation here differs strongly from that maintaining a *approximating domain*, where an uninitialized value must always be approximated by a $\top$ value.

the "load immediate" code and a report of the register in which the result is located. (Note that this is the same method used to generate code to handle initialized declarations described in the previous section.)

```
    public _IIntDomainInfo operatorAdd(_IIntDomainInfo argRHS) { return LocalBinaryCombineOrShor
tcut(this, argRHS, BinaryRegisterOp.AddOpType); }
    private _IIntDomainInfo LocalBinaryCombineOrShortcut(_IIntDomainInfo v1, _IIntDomainInfo v2,
 int opType)
    {

    _IGuidingIntDomainInfo intDomainMostRecentReturnFromGuidingDomain = (_IGuidingIntDomainInfo)
 this.Context().GuidingDomainGuidingInterface().GuidingValueIntReturnValue();

        if (((_IGuidingValueDomainInfo) intDomainMostRecentReturnFromGuidingDomain).FConcrete())
        {

    // ok if we know that the result of this operation is a concrete value, just generate the co
de to load it


    return (_IIntDomainInfo) this.CreateImmediateLoadValueOfSameType(intDomainMostRecentReturnFr
omGuidingDomain.SSValue());
        }
        else
            return(( _IIntDomainInfo) this.BinaryCombine((SCGValue) v1, (SCGValue) v2, opType));
    }
```

Figure 97: Handling Value Combination, Part I: Short-Circuiting..

In the event that the element returned by the approximating semantics for the current combination is *not* precisely known, the code for the method *LocalBinaryCombineOrShortcut* calls off to the method *BinaryCombine*. *BinaryCombine* creates the basic block necessary to combine the values present in each of the operands. Figure 98 shows the methods handling the emission of the residual code for a value combination.

The *BinaryCombine* routine (called by *LocalBinaryCombineOrShortcut* in Figure 97) returns a domain element of the appropriate type-specific class (either *IntSCGValue* or *PtrSCGValue*) by calling the factory method of the value being combined. The routine *GenericBinaryCombine*, however, performs the more significant work. This routine first obtains a new register in which the new calculated value will be held, and creates a new opcode sequence consisting of the two-operand opcode sequences appended together[49]. The code to compute this value is generated by emitting a final instruction that combines the register associated with the first value with the register associated with the second value using the appropriate register operator. The resulting sequence is handed back through the pseudo-pass-by-reference parameter, and the register in which the result is calculated is returned.

[49] Because of the "pseudo-register" naming scheme, the registers used in each block will not interfere.

```
    protected Register GenericBinaryCombine(SCGValue v1, SCGValue v2, int iOpcode, OpcodeSequenc
e s_opcodeSequenceOut[])
    {

    BinaryRegisterOp op = new BinaryRegisterOp(iOpcode, v1.RegisterResult(), v2.RegisterResult()
);

        // ok, first append the instruction to calculate this value to the code stream
        Registers.Free(v1.RegisterResult());
        Registers.Free(v2.RegisterResult());
        Register rOut = Registers.AllocateNewRegister();

        // ok, the code to calculate this value includes all of the code to calculate the
        //  operands, plus the code to perform the binary combination operation

        OpcodeSequence vectOpcodesNew = v1.m_opcodeSequence.Clone();

        vectOpcodesNew.Append(v2.m_opcodeSequence);
        vectOpcodesNew.Append(new RegisterMoveAndOpRTLInst(rOut, op));

        // ok, now return
        s_opcodeSequenceOut[0] = vectOpcodesNew;
        return(rOut);
    }
    private OpcodeSequence s_opcodeSequenceOut[] = new OpcodeSequence[1];
    protected _IValueDomainInfo BinaryCombine(SCGValue v1, SCGValue v2, int iOpcode)
    {
        Register r = this.GenericBinaryCombine(v1, v2, iOpcode, s_opcodeSequenceOut);

        return(v1.Factory(r, s_opcodeSequenceOut[0]));
    }
```

Figure 98: Handling Value Combination, Part II: Emitting Expression Calculation Code.

9.4.3.3 Handling the Least Upper Bound Operator

Figure 99 shows the implementation of the least upper bound operator for the code generation value domain. As was briefly mentioned in Section 9.4.2.1 and will be seen in further detail below, *writing* values to a location leads to the appending of the code to calculate them to the state's current basic block. In the current framework, least upper bounds are only taken between the contents of locations. The least upper bound operator should therefore never be applied to two non-bottom values. The code for *FiniteHeightLUB* makes use of this invariant in order to simplify its task.

```
    public _IValueDomainInfo FiniteHeightLUB(_IValueDomainInfo argRHS, boolean[] fChanged)
    {
        SCGValue narrowedValue = (SCGValue) argRHS;

        if (this.FIsBottomElt())
            return narrowedValue;
        else if (narrowedValue.FIsBottomElt())
            return(this);
        else
        {

   GlobalUtility.Assert(false, "any values being lubbed should have already been written => not
 in registers");
            // ok, both are just holding registers
            if (this.RegisterResult().FEquals(narrowedValue.RegisterResult()))
                return(this);
            else
                return FactoryMethodReturningTopElt();
```

```
    }
}
```

Figure 99: The Least Upper Bound Operator for the Code Generation Value Domain.

9.4.3.4 Conclusion

This section has examined the implementation of the value component of the code generation domains. The association of a basic block with each value allows for very straightforward implementation of value creation and combination routines, and seems to be a natural pairing indeed. The powerful capacity to perform *specialization* in conjunction with code generation comes about requires almost no additional effort. In particular, the code simply "peeks" at the return value for the approximating domain implementation of the current function. If the approximating domain result is a concrete value, code can be accumulated to take directly make use of that value rather than calculating the value anew at runtime.

9.4.4 State Operation Implementation

While conceptually simple, the implementation of the state domain for the code generation semantics involves considerably more mechanism than that of the value domain. Abstract states maintain both a directed graph of existing basic blocks and a distinguished "current basic block" at which code emission is taking place. The system is responsible for maintaining, merging and combining the specialized control flow graphs as well as for incorporating new fragments of the current basic block at those points at which locations are written.

9.4.4.1 Value Load/Stores

Recall that when values are stored into memory, the code associated with their calculation is placed into the state to which the write takes place; the stored values are then associated with the $\perp$ domain element. Figure 100 and Figure 102 show the implementation of a subset of the methods for reading and writing locations. In the interests of space, only half of the routines have been shown.

9.4.4.1.1 Handling Loads

Load routines create elements of this domain based on the contents of locations. Each element requires the specification of the code needed to load the variable into a register, and the identity of that target register. Figure 100 illustrates how two sample load methods establish these elements.

For loads from (non-escaped) variables, the code needed to perform the load consists merely of a load instruction from the appropriate stack location. The *ProcessVariableLoad* routine shown in Figure 100 illustrates the key code implementing a load from a variable location. The instruction performing the load

performs an indirect read on the offset of a frame pointer and writes the loaded value to a register. As mentioned in the context of the abstract state domain interfaces in Chapter 6, the compiler to the abstract semantics associates each variable in the program with a particular numeric identifier. The user's analysis state domains can deduce the offset of that variable in the stack by writing appropriate handlers to the routines *CreateVariableLValue* and *DestroyVariableLValue*.

For loads from memory locations other than program variables, the instructions in the returned domain element will include those necessary to calculate the location from which the load is taking place. In effect, translation of such an instruction is similar to the translation of a unary operator to the abstract semantics. The code necessary to perform this mapping can be seen in the *ProcessMemoryLoad* method in Figure 100. Here the code for the element resulting from the read first includes the code associated with calculating the address of the location (held in variable *narrowedLoc*) and then the instruction needed to perform the load itself.

```
    public _IIntDomainInfo ReadIntFilter(_IPtrDomainInfo loc, _IIntDomainInfo valueAlreadyRead,
int srcLoc) { return((_IIntDomainInfo) this.ProcessMemoryLoad(loc, (SCGValue) valueAlreadyRead))
; }

    public _IPtrDomainInfo ReadPtrFilter(_IPtrDomainInfo loc, _IPtrDomainInfo valueAlreadyRead,
int srcLoc) { return((_IPtrDomainInfo) this.ProcessMemoryLoad(loc, (SCGValue) valueAlreadyRead))
; }

    public _IIntDomainInfo operatorReadFromIntVariableFilter(int idVariable, _IIntDomainInfo vAl
readyRead, int srcLoc) { return((_IIntDomainInfo) this.ProcessVariableLoad(idVariable, (SCGValue
) vAlreadyRead)); }

    public _IPtrDomainInfo operatorReadFromPtrVariableFilter(int idVariable, _IPtrDomainInfo vAl
readyRead, int srcLoc) { return((_IPtrDomainInfo) this.ProcessVariableLoad(idVariable, (SCGValue
) vAlreadyRead)); }
        protected SCGValue ProcessMemoryLoad(_IPtrDomainInfo loc, SCGValue valueAlreadyRead)
        {
            Register rOut = Registers.AllocateNewRegister();
            PtrSCGValue narrowedLoc = (PtrSCGValue) loc;

            // ok, first copy in the code to calculate the arguments
            OpcodeSequence vectOpcodesNew = new OpcodeSequence();
            vectOpcodesNew.Append(narrowedLoc.CalculationOpcodes());
            vectOpcodesNew.Append(new LoadMemoryRTLInst(rOut, narrowedLoc.RegisterResult(), 0));

            return valueAlreadyRead.Factory(rOut, vectOpcodesNew);
        }

        protected SCGValue ProcessVariableLoad(int idVariable, SCGValue vInVariableLocation)
        {
            Register rOut = Registers.AllocateNewRegister();
            OpcodeSequence vectOpcodesNew = new OpcodeSequence();

    vectOpcodesNew.Append(new LoadMemoryRTLInst(rOut, Registers.FramePointer(), this.CBOffsetFor
IdVariable(idVariable)));

            return vInVariableLocation.Factory(rOut, vectOpcodesNew);
        }
```

Figure 100: Implementation of Value Load Methods.

```
        public int CBOffsetForIdVariable(int idVariable)
        {
            Object o = SCGStateDomain.this.m_MpVariableIdToIOffset.get(new Integer(idVariable));

    GlobalUtility.Assert(o != null, "No record of variable with id " + idVariable + " when looki
ng up in stack frame");

            return(((Integer) o).intValue());
        }
```

Figure 101: Code to Look up the Offset of a Variable in the Current Stack Frame.

9.4.4.1.2 Handling Stores

Methods to implement the storage of values to memory are conceptually similar to loads, but their handling differs in two important ways:

- **Additional Argument.** In addition to any location argument, storage routines must take the value to be written rather than returning it as an argument. Thus, the code necessary to perform this store must incorporate the code needed to calculate the value to be written. In effect, this makes implementation of storage to a variable similar to that needed for a unary operator and storage through a pointer similar to the handling of a binary operator.

- **Shift of Code from Value to State.** The value saved in memory following a store is treated as a bottom value in the code generation domain. At the point of writing, all code necessary to calculate that value and perform the store are transferred to the abstract state domain, and the stored value has no code and is held in no register.

These characteristics are reflected in the method implementations seen in Figure 102. All additions of code needed to necessary to compute the values used in the store and to perform the store itself are made to the state's current basic block rather than being accumulated in the value being stored itself. The value returned by the value filter is the bottom element in the code generation value domain.

```
    public _IIntDomainInfo WriteIntFilter(_IPtrDomainInfo loc, _IIntDomainInfo valueToBeWritten,
 int srcLoc){ return((_IIntDomainInfo) this.ProcessMemoryStore(loc, (SCGValue) valueToBeWritten)
); }

    public _IPtrDomainInfo WritePtrFilter(_IPtrDomainInfo loc, _IPtrDomainInfo valueToBeWritten,
 int srcLoc){ return((_IPtrDomainInfo) this.ProcessMemoryStore(loc, (SCGValue) valueToBeWritten)
); }

    public _IIntDomainInfo operatorAssignmentToIntVariableFilter(int idVariable, _IIntDomainInfo
 vCurrent, _IIntDomainInfo valueToBeWritten, int srcLoc) { return (_IIntDomainInfo) this.Process
VariableStore(idVariable, (SCGValue) valueToBeWritten); }

    public _IPtrDomainInfo operatorAssignmentToPtrVariableFilter(int idVariable, _IPtrDomainInfo
 vCurrent, _IPtrDomainInfo valueToBeWritten, int srcLoc) { return (_IPtrDomainInfo) this.Proces
sVariableStore(idVariable, (SCGValue) valueToBeWritten); }
        protected SCGValue ProcessVariableStore(int idVariable, SCGValue vToBeWritten)
```

```
        {
            // ok, first copy in the code to calculate the arguments
            SCGStateDomain.this.CurrentBasicBlock().Append(vToBeWritten.CalculationOpcodes());


    SCGStateDomain.this.CurrentBasicBlock().Append(new StoreMemoryRTLInst(vToBeWritten.RegisterR
esult(), Registers.FramePointer(), this.CBOffsetForIdVariable(idVariable)));
            return vToBeWritten.Factory(false, true);
        }
        protected SCGValue ProcessMemoryStore(_IPtrDomainInfo loc, SCGValue vToBeWritten)
        {
            Register rOut = Registers.AllocateNewRegister();
            PtrSCGValue narrowedLoc = (PtrSCGValue) loc;

            // ok, first copy in the code to calculate the arguments
            SCGStateDomain.this.CurrentBasicBlock().Append(narrowedLoc.CalculationOpcodes());
            SCGStateDomain.this.CurrentBasicBlock().Append(vToBeWritten.CalculationOpcodes());


    SCGStateDomain.this.CurrentBasicBlock().Append(new StoreMemoryRTLInst(rOut, narrowedLoc.Regi
sterResult(), 0));

    return vToBeWritten.Factory(false, true);  // returns the bottom element in the domain
        }
```

Figure 102: Implementation of Value Store Methods.

9.4.4.1.3 Conclusion

Loads and stores represent the coupling of the abstract state and abstract value domains, and are the locus of important machinery in the code generation domain. Particularly important in this respect is the availability of "filter" routines in the state interface. These routines allow state domains to perform custom actions at the points at which variables and memory locations are read and written. Placing such filters in the state rather than the value domain interfaces facilitates the custom handling of reads and writes by the states. This is achieved without adversely impacting the implementation of domains in which the action of such "filter" routines is limited to value manipulation.

9.4.4.2 Splitting a State

As originally discussed in Chapter 6, the presence of unknown predicates during analysis can requiring the splitting the path of analysis to conservatively approximate all possible routes of execution through the program. The customized behavior of abstract state domains upon encountering such splits is encapsulated in the *DynamicSplit* method of the abstract state domain. Execution of this routine results in the establishment of two states representing each fork of the split: A state that will continue on beyond the function call, and the state to be returned by the method. In order to allow for sharpening of knowledge concerning the values in the current state, the method takes as an argument a predicate value. This value is known to be true for the abstract state that continues, and known to be *false* for the other branch (i.e. for the abstract state returned by the function)

The implementation of the DynamicSplit routine for the code generation domain is shown in Figure 103. The routine first creates the instructions necessary to calculate and evaluate the predicate, as passed in via the value *vCalculated.* Following the formulation of these instructions, an instruction is appended which compares the value of the predicate to zero. If the predicate is zero, the branch will be made. Otherwise (if the predicate is true), the path of execution will continue to the next state.

Because no epistemic sharpening is performed in this domain based on knowledge of the predicate, the other state to be returned is simply a clone of the current state. "Backpatch" requests are queued up for both the split state and the subsequent state. Each such request asks for the branch instruction (and the subsequent fall-through instruction) to be patched with the subsequent address taken on by each fork of the split. Thus, when execution reaches the beginning of each fork, that fork will be responsible for backpatching the branch instruction that points to it. Finally, the implementation of the dynamic split creates a new basic block (representing the *true* side of the conditional) and backpatches the "fall through" instruction following the branch with the code emission address associated with the new basic block. This newly created basic block will be the current basic block for subsequent states.

```
        public _IStateDomainInfo DynamicSplit(_IIntDomainInfo vCalculated)
        {
            IntSCGValue  vCalculatedNarrowed = (IntSCGValue) vCalculated;
            Register rCompareImmediate = Registers.AllocateNewRegister();
            // note that this contains a backpatch slot!
            // ok, we branch if the predicate is NOT true => does  equal 0  at runtime
    SCGStateDomain.this.CurrentBasicBlock().Append(vCalculatedNarrowed.CalculationOpcodes());

    SCGStateDomain.this.CurrentBasicBlock().Append(new BranchIfZeroRTLInst(null, vCalculatedNarr
owed.RegisterResult()));

            // ok, now that the code is generated, now prepare the new states.
            //  each of these states will be associated with backpatching machinery

    // ok, the split state represents the case where the predicate is NOT equal to the specified
 value
            // calling this directly allows both copying the graph and resolving  what
            // the current block is
            SCGStateDomain stateDomainSplit = (SCGStateDomain) SCGStateDomain.this.Clone();
            // the split state represents the branch case

    stateDomainSplit.AddBackpatchLocation(new BackpatchLocation(SCGStateDomain.this.CurrentBasic
Block(), false));
            // the current state DOES represent the fall-
through case.  here, the predicate does have the
            //  specified value at runtime

    SCGStateDomain.this.AddBackpatchLocation(new BackpatchLocation(SCGStateDomain.this.CurrentBa
sicBlock(), true));
            SCGStateDomain.this.EstablishNewBasicBlockAndPerformBackpatching(-1, false);

            // ok, return the split state
            return(stateDomainSplit);
        }
```

Figure 103: Implementation of the Split Method for the Code Generation Domain

9.4.4.3 *Joining States*

The section above discussed the code used to split a state. This section examines the converse operation: Joining two preexisting states with the least upper bound operator. As discussed in Chapter 6, there are two different forms of least upper bound between states. The first and most common type of least upper bound is a least upper bound performed from a saved-away state *to* the current state. A second form of least upper bound is that performed from the current state to a saved-away state. The result of this least upper bound is then saved away and continues on as the current state.

By definition, least upper bound of two abstract state domains *A* and *B* must return an abstract state domain *A* and *B* which soundly approximates both *A* and *B*. Recall that the notion of "approximation" used for elements of the code generation abstract state domain is very similar to that of graph containment. Thus, we will expect that result of performing either of the two varieties of least upper bound operators on two abstract states would be to create a new state that contains each of the argument states as a subgraph.

Given this general understanding of the requirements of implementing the least upper bound operator, the implementation of each of these two types of operations is discussed below:

9.4.4.3.1 *Least Upper Bound From Current State*

Because they both implement central control-flow operations, the two handlers have many low-level mechanisms in common. The interface method implemented in Figure 104 is *FiniteHeightLUBFromCurrentStateAndSetAsCurrentState*. This routine takes the least upper bound of the current state with the left hand side (the object on which the method is invoked), and sets the result as the new current state. A snapshot of the resulting state is also returned, to allow for storing away for later accumulation or for instantiation.[50] The code for this routine delegates to the routine *FPerformLUB* within the current state, also shown in Figure 104.

The implementation of the *FPerformLUB* method for the code generation domain first sets up requests to backpatch the current basic block in each of the states being joined. The routine then takes the least upper bound of the generated code graphs associated with each state being combined. This is an involved operation not shown here; the interested reader is referred to the complete code for this domain at the URL referred to in Appendix 3. (In essence, it involves joining the two control flow graphs together, guaranteeing that information for shared nodes is merged and that more recent versions of nodes produced during fixed point iteration take primacy over earlier versions of nodes. Finally, a new basic block is

[50] The ability to store away the resulting state is particularly important for the implementation of fixed-points in loops (recursive and otherwise), where a loop entry approximation accumulates each state that successively reaches the top of the construct.

established for accumulation of code in subsequent continuation of execution in this state, and backpatches are made to the two states that were joined.)

Upon returning, the code for *FiniteHeightLUBFromCurrentStateAndSetAsCurrentState* simply returns a clone of the current state, which can then be used for future accumulation. Note that because all backpatching has been performed for this state, the clone will not need to duplicate any backpatch requests.

```
    public _IStateDomainInfo FiniteHeightLUBFromCurrentStateAndSetAsCurrentState(_IStateDomainIn
fo domainStateCurrAndLHS, int srcLoc, boolean fChangedOut[])
    {
        // ok, we want to
        //      LUB together the graphs associated with each of the states
        //      this gives us a single graph that approximates either of the component graphs
        //      start execution at the resulting graph
        //      do any necessary backpatching
        //      start executing

        SCGStateDomain narrowedDomainStateCurrAndLHS = ((SCGStateDomain) domainStateCurrAndLHS);

    fChangedOut[0] = narrowedDomainStateCurrAndLHS.m_dummyToCurrent.FPerformLUB(this, srcLoc, tr
ue);
        SCGStateDomain clone = narrowedDomainStateCurrAndLHS.NarrowedClone();
        this.PerformSetAsCurrentState(srcLoc, true);
        return clone;
    }
    boolean FPerformLUB(SCGStateDomain domainNonCurrent, int srcLoc, boolean fEnforceFiniteHeigh
t)
    {
        GeneratedCodeGraph graphNonCurrent = domainNonCurrent.m_graph;

    // ok, make it so there are FALL THEOUGH patches from both of the old bb's being LUBbed to t
he current BB

    SCGStateDomain.this.AddBackpatchLocation(new BackpatchLocation(domainNonCurrent.CurrentBasic
Block(), true));

    SCGStateDomain.this.AddBackpatchLocation(new BackpatchLocation(SCGStateDomain.this.CurrentBa
sicBlock(), true));

        boolean fChanged = SCGStateDomain.this.m_graph.FChangedLUBTo(graphNonCurrent);
        // note that ordering is important here -- we want to inform of any finite height
        //  restrictions prior to creating a new block (just in case creating a new block will
        //  actually require reuse of an existing, canonical block)

        EstablishNewBasicBlockAndPerformBackpatching(srcLoc, fEnforceFiniteHeight);
        return fChanged;
    }
```

Figure 104: Machinery to Perform a Least Upper Bound and Accumulation from a Current State and to a Non-Current State in the Code Generation Semantics.

9.4.4.3.2 Least Upper Bound To Current State

The routine *LUBToCurrentState* shown in Figure 105 illustrates the handling of a request to calculate the least upper bound of a saved-away state (the object on which the method is invoked) with the current state (passed in through the argument *domainStateCurrAndLHS*). Although it requires only a piece of its functionality, this routine makes use of the workhorse method *FPerformLUB* that was shown in Figure 104

and which formed a central part of the handling of the method *FiniteHeightLUBFromCurrentStateAndSetAsCurrentState*. In particular, the routine calls off to the *FPerformLUB* routine with a final argument of *false*. This argument indicates that no finite height for the domain need be enforced. The return value from *FPerformLUB* is ignored, since *LUBToCurrentState* does not need to return any indication as to whether the result is higher in the lattice than one of the arguments.

```
public void LUBToCurrentState(_IStateDomainInfo domainStateCurrAndLHS, int srcLoc)
{

((SCGStateDomain) domainStateCurrAndLHS).m_dummyToCurrent.FPerformLUB(this, srcLoc, false);
}
```

Figure 105: Implementation of the Least Upper Bound of a Saved-Away State to the Current State.

9.4.4.3.3 Summary

This section has briefly examined the semantic rules by which control flow joins are merged. The implementation of the two types of joins have been examined – a join *to* the current state and a join occurring *from* the current state to a saved-away state that enforces a finite height for the domain. Both implementations made use of the same internal machinery that backpatched the "fall through" jump slots of each of the abstract domains being joined so as to point to the new current state. The control flow graph of the resulting domain element was the least upper bound of the control flow graphs of the domains being joined. This graph combination is performed by the *GeneratedCodeGraph* class, which is not shown here. (See the code referred to in Appendix 3 for a listing of this class). This class creates a code graph approximating each of the code graphs being joined. This is accomplished by maintaining a single shared node for nodes held in common between the argument graphs, superimposing the graph edges for cases in which each of the graphs being approximated contains different edges, and giving priority to edges and nodes resulting from later iterations of a fixed-point iteration.

9.4.4.4 Stack Maintenance

Another important function within the code generation state semantics relates to the collection of information provided by the analysis machinery through stack operation notifications. Figure 106 shows the code implementing the stack-related method *CreateVariableLValue* and *DestroyVariableLValue*.

Figure 106 shows the routine used to handle notifications of adjustments to the run-time stack pointer that occur to accommodate local variables. The code generation approximating state domain uses this information in two ways: To emit code for run-time stack adjustments, and to collect information on the placement of variables within the current stack frame.

The first task undertaken by the domain is the emission of the code needed to perform the appropriate stack adjustments at runtime. As suggested by the routine shown in Figure 107, the code to be emitted is very simple and uniform. When notified of the *creation* of a variable in the abstract semantics, the state domain emits code to add to the stack pointer an offset corresponding to the size of the variable. Conversely, when the abstract semantics interpreter destroys a variable location, code is emitted to perform a comparable action at runtime, by subtracting the appropriate offset from the stack pointer. Note that because this code is emitted incrementally, it can yield a relatively inefficient contiguous sequence of stack adjustments (e.g. simply adding constant values to the stack pointer for many instructions in a row). A more efficient strategy would allow for "batching" such adjustments until the first subsequent code is emitted.

As noted above, the information provided through the parameters of this routine is used not only to emit code used to adjust the stack pointer at run-time, but also to keep track of stack structure. Recall from the discussion in Section 9.4.4.1 that loads and stores from variables make use of the routine *CBOffsetForIdVariable* (shown in Figure 101) to determine the offset for a specified variable id. The code for *CreateVariableLValue* and *DestroyVariableLValue* maintain the bookkeeping data structures used to keep track of these offsets – in particular, the table *m_MpVariableIdToIOffset* of the *SCGStateDomain* class. (A more general approach would maintain a scope data structure that contains information on each variable in the current scope.)

```
        public void CreateVariableLValue(int idVariable, int cbSize)
        {
            EmitAdjustStackPointerCode(cbSize);
            // record the offset for the specified variable
            Integer refKey = new Integer(idVariable);


    SCGStateDomain.this.m_MpVariableIdToIOffset.put(refKey, new Integer(m_iNextVariableOffset));
            m_iNextVariableOffset += cbSize;
            SCGStateDomain.this.m_MpVariableIdToCBSize.put(refKey, new Integer(cbSize));
        }
        public void DestroyVariableLValue(int idVariable)
        {
            Integer refKey = new Integer(idVariable);

    Integer sizeObject = (Integer) (SCGStateDomain.this.m_MpVariableIdToCBSize.get(refKey));

    GlobalUtility.Assert(sizeObject != null, "No record of variable with id " + idVariable + " i
n stack frame -- do not know variable size in DestroyVariableLValue");
            EmitAdjustStackPointerCode(-1 * sizeObject.intValue());
            // ok, now remove all records of this variable from the current scope
            SCGStateDomain.this.m_MpVariableIdToCBSize.remove(refKey);
            SCGStateDomain.this.m_MpVariableIdToIOffset.remove(refKey);
        }
```

Figure 106: Implementation of Routines to Emulate and Monitor Stack Manipulation.

```
        private void EmitAdjustStackPointerCode(int cbSizeInBytes)
        {
            int opType;
            if (cbSizeInBytes > 0)
                opType = BinaryRegisterOp.AddOpType;
            else
                opType = BinaryRegisterOp.SubOpType;

            Register rCountBytes = Registers.AllocateNewRegister();

    SCGStateDomain.this.CurrentBasicBlock().Append(new LoadImmediateRTLInst(cbSizeInBytes, rCoun
tBytes));
            Registers.Free(rCountBytes);

    SCGStateDomain.this.CurrentBasicBlock().Append(new RegisterMoveAndOpRTLInst(Registers.FrameP
ointer(), new BinaryRegisterOp(opType, Registers.FramePointer(), rCountBytes)));
        }
```

Figure 107: Implementation of Routine to Emit Code to Adjust Stack Pointer

9.4.5 Program Specialization and Run-time Code Generation

9.4.5.1 Specialization

Section 9.4.3.2 and Figure 97 demonstrated how a few lines of code within the routine handling value combination allows the code generation domain to emit efficient code for expressions in which the approximating domain evaluates to a constant. In such cases, rather than emitting code to perform a general-purpose evaluation of the expression, the domain can simply emit specialized code to load the appropriate concrete value into a register. Similar semantics are implicitly supported in control flow as well, because the analysis system only invokes a dynamic split handler for those cases in which the truth status of the splitting predicate is *not* known. For cases in which the predicate is associated with a known truth-value, only the appropriate branch of execution is followed. In effect, the system executes as much of the program as possible (or desirable) at the time of analysis, and emits the remainder for execution at run-time. Shifting "doable" work to analysis time allows the code emitted by the system to be considerably more efficient, by virtue of *specialization* to the particular values known to circulate in the program at runtime. These performance improvements are made possible at the most fundamental level by the presence of static information concerning the execution of the program being analyzed.

Because of its capacity to perform such optimizations, the execution of the code generation domain over a program *P* during analysis can be viewed as conducting a *specialization* of *P*.

Program specialization is a program transformation technique that accepts as input a piece of code and a specification of a subset of the external inputs to that code. The program specializer outputs a specialized (or "residual") piece of code that has the same functionality as the original code, but exploits knowledge of the specified inputs to safely exploit optimized special cases where the original program made use of general, abstract mechanisms. Program specialization offers an *automatic* means of safely attaining the

performance advantages of "hard coding" special cases without compromising the generality and abstract character of the original code. In many instances, the performance advantages gained through specialization are substantial, and can involve the elimination of complicated general-purpose mechanisms where they are not needed and the shifting of substantial computations from the time at which the code is used to specialization-time.

Slightly more formally, suppose we have a program $p(a,b)$: $Value \times Value \rightarrow Value$ taking formal parameters a and b, and an interpreter I: $String \times Values \rightarrow Value$ for the language in which p is written. For any actual parameters x and y, a program specializer *mix*[51] is defined by the relation.

$$mix(p, x) = p_x$$

where

$$I(p_x, y) = I(p, (x, y))$$

Here, p_x represents the "specialized program" output by applying the program specializer *mix* to program p with actual argument x. More intuitively, p_x represents the program p specialized to a specific value for one of its input parameters (and possibly any other static data available inside p). The last equation simply establishes a correctness (consistency) relationship between p_x and p itself, requiring that p_x run compatibly with what would be expected from p given a first parameter of x.

9.4.5.2 The Notion of a Specialized Specializer

The fact that analysis using the code generation domain conducts a *specialization* of a program to the knowledge that is statically available in the program text constitutes an interesting observation, and one that testimony to the generality of the analysis interfaces presented here.

The fact that the code generation domain is performed within the context of a "compiled analysis", however, adds in an additional twist to the system. In generic program specialization (much as in generic compilation), a general-purpose program specializer *mix* takes as input a program P and some partially specified input i to P, and outputs a specialized program P_i representing P specialized to operate on i.

While analysis using the code generation domain does perform program specialization, here there is no need to specify P – in a very real sense P is "built in" to the analysis. The use of the code generation domain within a "compiled analysis" framework in effect represents application of a *specialized* program specializer

[51] This is a historical name standing for "mixed computation" – a computation taking place partly at the time of specialization and partly at a later point.

mix_P. Said another way, the specialization performed by the code generation domain is performed by a very particular type of specializer – it is performed by a specializer that is itself specialized to operate on the particular program being analyzed! The next section examines a potentially valuable implication of this observation.

9.4.5.3 The Application of Specialized Specializers in Run-Time Code Generation

9.4.5.3.1 Background

The fact that the code generated by the domain is produced by means of a "specialized specializer" is not merely an intellectual curiosity: It has some important applications as well. Beginning with the observation in [Massalin, 1992 #97] regarding the fundamental similarity between run-time code generation and partial evaluation, researchers have fruitfully applied techniques from program specialization to problems in run-time code generation.

Perhaps surprisingly, the notion of "program specialization" can be directly applied to the process of generating code at runtime. Here p represents some piece of code which we wish to specialize at run-time to particular "specialization-time static" values x that are known at that point, while leaving other values y free to vary. (For example, to accelerate the inner loop of a matrix multiply routine, we may wish to specialize a dot product routine for the left-hand parameter being fixed to some known row of the left hand matrix, while the right hand varies among different columns of the right hand matrix). The specialization process *mix(p, x)* here takes place at run-time. This specialization emits a new piece of object code p_x that is specialized to the particular x but can be subsequently run many times with different values y (e.g. different columns of the right hand matrix). Thus, we can think of many attractive examples run-time code generation as *specializations* of code to some known partially values, where the specializations happen to take place at run-time.

The importance of specialization to run-time code generation, however, goes beyond its application to the serving as an informal source of ideas and techniques for the emission of optimized code. In particular, a formal understanding of program specialization has provided researchers with concrete ideas for constructing *efficient code generators*. In the model above, we thought of run-time code generation itself as the run-time application of a program specializer *mix* to a particular piece of code p using some knowledge about a particular set of "specialization-time static" information x. This is a convenient model, but offers no assurances that the specialization itself (i.e. the emission of the specialized code p_x) will take place quickly. What we are seeking is a *rapid* way to produce p_x given the knowledge of p and input x.

Fortunately, specialization can help us here as well. The key realization is that while the input x will typically not be known until the point during run-time at which the specialization is required, the code p will typically be known *statically,* as will knowledge that it is against x (*i.e.* particular row vectors) that we will wish to specialize p. For example, at development-time we will be able to anticipate that we wish to specialize a *particular* function to a series of different particular arguments over time. Rather than making use of a fully general *mix* to specialize our code p at run-time through the application *mix(p, x)*, we can exploit the staged nature of p and x to craft a customized, high-performance specializer for p at development-time. In other words, we will create a *specialized* version of *mix* (call it $mix_{p,1}$) is "hardwired" to specialize p to *whatever* "specialization-time constants" x are provided at run-time.[52] Providing a particular specialization-time constant x to this "specialized specializer" (or "generating extension") $mix_{p,1}$ at run-time as $mix_{p,1}(x)$ will result in the rapid emission of a p_x – the specialized version of p which can be combined with different values y.

Coming back to the particular issue at hand, we can recognize that use of the code generation domain in conjunction within a "compiled analysis" implements $mix_{p,1}$. That is, when a compiled analysis making use of the code generation domain is provided with partial information with which to specialize (any inputs x specified by the user during analysis and the static information internal to a program), it will emit as assembly code a version p_x of program p that is specialized to those inputs. The formal equivalence of these approaches raises the opportunity of using the code generation domain (or a variant thereof that emits machine code) to perform run-time code generation.

The close affinity of specialization and run-time code generation is attested to by its use within an increasing number of studies. Two of the most important works in this area are [Grant, Mock et al. 1997] (which uses run-time code generation to specialize regions of a variety of granularities to particular values), and [Noel, 1996 #119], which specializes functions to particular arguments at runtime. Other well-known papers describe the specialization of functions to the particular types being manipulated [Chambers, 1989 #118], specialize compressed programs to particular architectures [Ernst, 1997 #123], operating system structures to particular contexts [Massalin, 1992 #97], and graphics routines to particular relationships between source and destination rectangles [Locanthi, 1987 #120].

[52] Note that the subscript "1" indicates the particular formal parameter with respect to which this specializer will specialize when an argument is provided

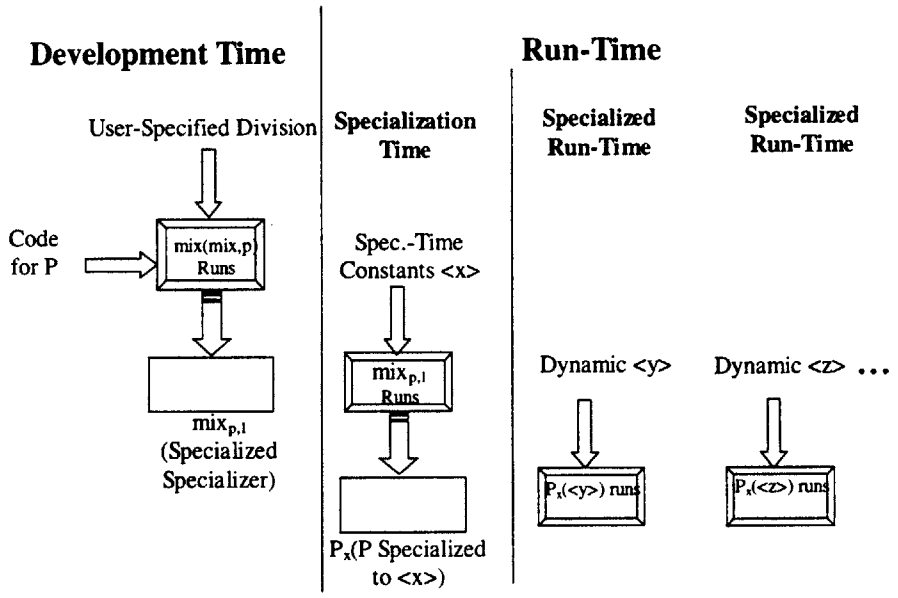

Figure 108: The Use of a "Specialized Specializer" for Run-Time Code Generation

9.4.5.3.2 Prospects for Use

The sections above established the formally applicability of the code generation domain for performing efficient run-time code generation. Figure 109 illustrates how the code generation domain would be used in run-time code generation. While time constraints prevented this thesis from investigating these possibilities in greater depth, two considerations regarding the efficiency of the analysis/code generation process suggest that despite its apparent applicability, TACHYON may not be well suited to application in this area.

Before beginning the discussion, it may be worth recalling the central importance of efficiency in run-time code generation. While the time required for *static* analysis and code generation delays only the developer and does not directly affect the end user, all analysis and optimization performed at run-time immediately impacts the overall running-time of the application. In order to have a net positive impact on the performance of the application, all time spent *generating* code must be recouped by the time saved due to the greater efficiency of the emitted code. Because generating higher-quality code typically requires more costly analysis and optimization, performing effective run-time code generation requires care and balance. In particular, a system performing run-time code generation must strike an acceptable balance between too little analysis/optimization (yielding poor quality emitted code that has poor performance) and too much analysis/optimization (in which case the emitted code is highly efficient, but is incapable of fully amortizing the time spent generating it). In the context of the code generation domain, the danger is clearly on the side of performing too much analysis/optimization.

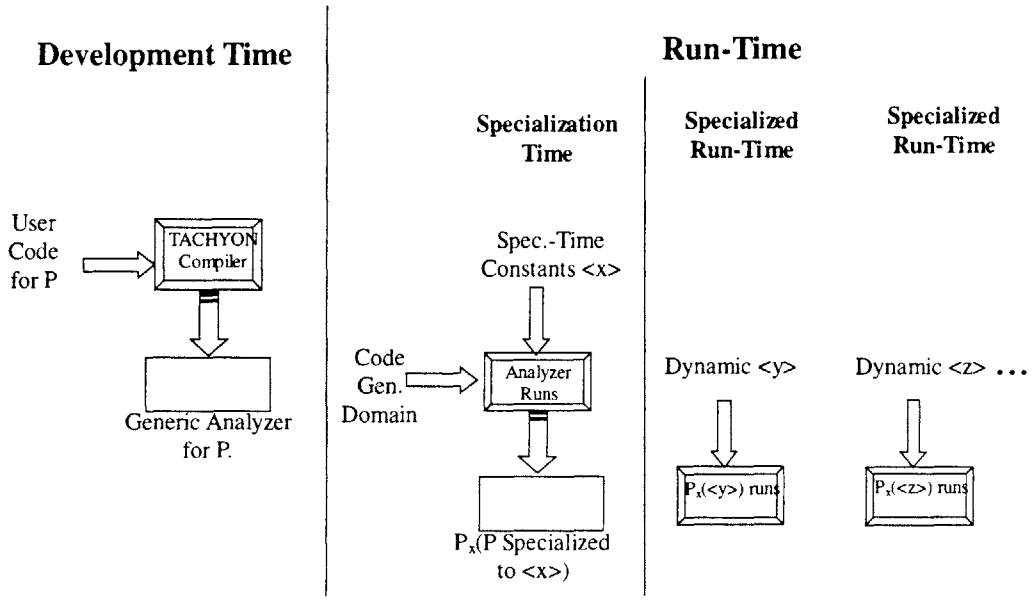

Figure 109: The Use of a Compiled Generic Analysis as a Specialized Specializer in Code Generation

TACHYON has two central efficiency disadvantages relative to other run-time code generators making use of specialized specialization (e.g. [Grant, Mock et al. 1997] and [Noel, Hornof et al. 1996]) The next two sections discuss each of these shortcomings.

9.4.5.3.2.1 ONLINE VS. OFFLINE TRADEOFFS

All specialization-based run-time code generation systems built thus far ([Grant, Mock et al. 1997], [Noel, Hornof et al. 1996], [Leone and Lee 1996]) make use of what is known as "offline analysis" in order to determine what information within a program can be statically known and which will not be known until run-time. Offline systems are traditionally divided into two phases. The specializer first conducts a rapid *binding-time analysis* (BTA) phase which attempts to approximately detect which values in a program are statically known and which are dynamic. The binding time analysis is followed by a (potentially much lengthier) specialization phase, which performs all operations recognized as static. In effect, in such systems the program *mix* is thoroughly specialized *both* to knowledge of the program p itself and to knowledge as to *which of the parameters of p will be considered variable and which will be viewed as fixed.* (Indeed, we implicitly hinted to the applicability of this approach above by denoting the specialized specializer as $mix_{p,1}$.) This convenient staging in offline analysis methods permits a compiler to exploit the fact that the program p and the static/dynamic classification of each piece of the input are often known

statically, and to apply the BTA phase of the analysis to the program *statically* rather than at run-time. Thus, the code for $mix_{p,I}$ takes advantage not only of knowledge of p, but of knowledge of the fact that it is the formal parameter x that will be known. The static/dynamic classification of expressions in p is thus "hardcoded" in the specialized specializer $mix_{p,I}$.

In the offline methodology, then, the specialization system $mix_{p,I}$ has *built-in* knowledge of which values and portions of the computation will be specialized and which will be residual and have to be emitted as code. This has one major advantage and two major disadvantages.

The advantage is that the specialization process can itself be executed very efficiently. The disadvantages are twofold:

- Because the issue of which expressions will be known and unknown must be decided prior to knowing the actual data, not all opportunities for specialization are recognized. Figure 112 and Figure 113 illustrate trivial examples of this problem.

- The need to specialize mix to *both* the program p and the static/dynamic classification of the input data means that different "specialized specializers" will be needed for contexts in which different variables are known or unknown. In terms of the notation introduced above, there must be different $mix_{p,n}$ constructed for each situation in which *different* sets of parameters (x or y in the example above) are known.

```
static1 = 10   // abstracted to "static"
…
if (static1 > 0) // BTA knows this is static but not the value
   v = 20;
else
   // BTA views else clause as possible ⇒ v judged "dynamic"
   v= dynamic1;
```

Figure 110: Demonstrating the Approximative Character of BTA's Static/Dynamic Classification Algorithm: Case I.

```
// here, v is judged "dynamic" even though for static1=0 it will be static.
v = dynamic1 * static1 = 10
```

Figure 111: Demonstrating the Approximative Character of BTA's Static/Dynamic Classification Algorithm: Case II.

By contrast, an online analysis and specialization system is more flexible, but pays for that flexibility in performance. Within online systems, no attempt is made to specialize *mix* to a particular division between what is known and what is unknown. Rather than making use of $mix_{p,I}$ such systems would instead use

mix_p. In this approach, the system takes the partial input specification (e.g. "x=x" or "y=y") and determines "on the fly" which expressions and variables are known at which points in time, and which cannot be known until run-time. Offline systems therefore require the specialization machinery in a specializer mix_p to perform a much more substantial – and time-expensive – analysis at run-time. This form of analysis has complementary strengths and weaknesses to those seen above for offline systems. The major disadvantage of the online techniques is the use of a more heavy-duty set of specialization machinery. However, the full-bodied analysis can discover important opportunities for specialization not discovered by offline systems (e.g. discover that v is definitively *static* in Figure 112 and is static for certain values of *static1* in Figure 113). The online approach also allows a single specializer to be used in a variety of specialization contexts.

9.4.5.3.2.2 INTERPRETIVE OVERHEAD

An additional source of performance overhead for the analysis system arises from the extensive use of interface polymorphism needed to maintain extensibility. As discussed in Chapter 3, while the analysis machinery is specialized to the program to be analyzed, lacks the traditional advantages of specialization to the semantics rules associated with the particular analyses being performed. As a result, considerable implicit interpretive overhead is sustained in the run-time binding of interface method calls to value and state domains, as well as in the *AbstractValue* and *AbstractState* machinery that pairs up information drawn from identical domains. This interpretive overhead places the efficiency of a hypothesized run-time code generation domain at a serious disadvantage relative to more specialized run-time code generation systems such as [Grant, Mock et al. 1997] or [Noel, Hornof et al. 1996]. More importantly, the need to guarantee that the abstract execution is performed soundly for *any* legal parameterized domain makes TACHYON's analysis much slower than that of other contemporary abstract interpretation systems. [Chen and Harrison 1992; Ayers 1993]

9.4.5.3.2.3 CONCLUSIONS

TACHYON performs a very heavyweight form of analysis; this yields both more precision in analysis but also much higher running – and thus code-generation – times. The trend in run-time code generation systems is towards *lighter-weight* code generation systems that still perform a strong core set of optimizations, rather than towards incrementally more expensive systems that open opportunities for performing more specializations. (For example, see[Grant, Mock et al. 1997], [Noel, Hornof et al. 1996], [Leone and Lee 1996], and [Engler 1996]) While the stronger analysis and optimization that accompanies online analysis is most definitely desirable in certain contexts (such as cases in which the residual code is guaranteed to be executed hundreds or thousands of times), in most cases it will be more of a liability than an asset. On the other hand, the very substantial interpreted overhead that arises from extensibility has no

redeeming value for run-time code generation, and makes TACHYON uncompetitive relative to other state-of-the-art run-time code generation systems.

If the current system were to further allow for thorough specialization of the analysis system with respect to the semantic domains involved, the TACHYON framework would allow for a very interesting experiment in run-time code generation: The first online run-time code generation system. While it is unlikely that this system would be acceptable as a standalone solution for run-time code generation, it may well be that a hybridization of the online and offline approach could offer attractive performance at both analysis time and code generation time.

9.4.6 Conclusion

This section has provided a brief overview of the code generation domain – the largest and most complicated semantic domain yet implemented in the TACHYON runtime. Because of the substantial size of the implementation, significant portions have been omitted from the exposition. Most importantly, the discussion has omitted an concrete exposition of how the system maintains the control flow graphs that actually store the code to be emitted. The implementation of these graphs was somewhat challenging but relatively straightforward. Interested readers are referred to the code referenced in Appendix 3. The code that was presented above should help to demonstrate the rather surprisingly natural fit of code generation into the abstract execution framework. As is the case for the other domains, taking this perspective on code generation allows the code generation domain to take advantage of the machinery that comes automatically with the surrounding framework. Doing so greatly lessons the amount of effort needed to implement a code generator – much less a *specializing* code generator. For the code generation domain, this has two major advantages:

- **Ease of implementation.** Making implicit use of the analysis framework to guide the code generation process allows the code generation to be accomplished with far less complexity and code than would be required in a separate implementation.

- **Higher Efficiency.** As was discussed in Section 9.4.3.2, placing the code generation domain into an extensible abstract execution framework allows it to make use of the computations of the approximating domain with just a few lines of code (see in particular Figure 97). Most importantly, with the choice of a high-precision approximating domain (such as those seen in Chapter 12) this has the effect of turning the code generation process into a very high-quality program specializer. In effect, the analysis machinery takes care of performing as much computation within the program as the user desires to

perform at runtime and for which information is available. The remainder of the program is output as efficient residual code, to be executed later at runtime in order to complete the system's operation.

9.5 Conclusion

This chapter has discussed a few of the example collecting domains implemented within the TACHYON framework. The domains demonstrate both the relative ease with which new analyses can be implemented as well as the generality of the interfaces. Of particular note in both of these regards was the implementation of the code generation domain. By implementing code generation in an abstract execution framework and letting it "ride" on top of results returned for operations in the approximating value domain, we were able to create a strongly optimizing code generator. While this domain offers tantalizing possibilities for performing the first experiments in online run-time code generation, it seems likely that the relatively slow performance of compiled analyses renders this approach impractical.

Chapter 10 Example Approximating Value Domains, Part I – Domains with Fixed Structure

The last chapter described a few examples of collecting semantic domains. These domains accumulate information on program behavior, but do not affect the set of concrete states approximated during analysis. By contrast, this and the following chapter present several example *approximating value* domains. The function of these domains is to approximate run-time values, and the choice of these and approximating state domains directly determines the set of concrete states that are simulated during analysis.

All of the domains presented in this and the following chapter approximate integer values, although the distinctions they are capable of capturing differ from domain to domain. As such, the two chapters should be taken as a logical unit. Space considerations have forced the creation of two separate chapters, each somewhat different in emphasis.

This chapter presents the first two example domains, with fixed lattices – the expressiveness of the domains is "hard wired" into the code to implement them. The first domain is a "toy" domain created for pedagogy rather than for practical use. This domain is used to illustrate the general manner in which approximating semantic domains are formulated, and to provide a basis for understanding and comparing with other domains. The second domain is a popular abstract value domain used in the literature. While nearly as simple as the toy domain, this domain offers potential for practical use in certain circumstances.

By contrast, the approximating value domains discussed in the subsequent chapter are *extensible* domains, in the sense that they can be used to implement values of a wide range of precision. These domains offer the user finer control over the analysis running time, and are not merely interesting applications of the existing framework, but also constitute noteworthy representations in their own right.

This chapter will follow the pattern set in the examination of collecting domains, where we examine the implementation of the semantic rules for a small set of methods. The next chapter will examine some issues in additional detail.

10.1 Even/Odd Approximating Domains

The first approximating value domain that we will examine is a toy domain that approximates values as either even, odd, or unknown. Figure 112 shows the lattice for this domain. Characteristic of a "diamond"

domain, values are classified as either "definitively even", "definitively odd", "either odd or even", "neither odd nor even".

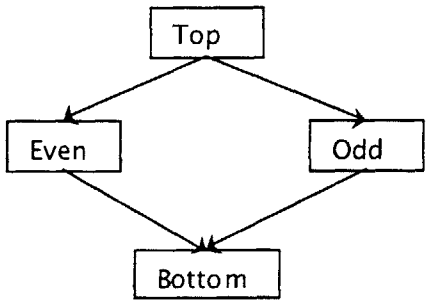

Figure 112: The Topology of the Abstract Domain Lattice Associated with the Even/Odd Abstract Value Domain.

It is worth remarking on the close resemblance between the lattice in Figure 112 and that seen in the previous chapter in the context of *collecting* domains (e.g. for the Initialization Status domain). While the lattice topology is similar, it is important to stress they describe very different mathematical domains: While approximating domains approximate particular values (e.g. a value known to be even), collecting value domains represent abstractions of value *sequences*. This distinction implies much of the behavior that may otherwise seem puzzling: For example, a *approximating* semantics representation of an uninitialized value will *always* be associated top domain element in the appropriate lattice and thus serve as a sound approximation to any other domain element from the same lattice. By contrast, this is far from always being true in the *collecting* semantics domain: For example, in the Initialization Status domain, the element arising from the abstraction of an uninitialized value sequence should not be and is *not* a legal approximation to the abstraction of *any* initialized value sequence.

10.1.1 Domain Element Creation

Figure 113 shows the routines used to create new domain elements from statements creating values in the code. As noted above, while collecting domains need not associate top domain elements with uninitialized values, approximating domains must. Thus, the code for *CreateUninitialized* (which handles creation of values with an uninitialized values) yields a top element within the domain. The routine *CreateFromConstant* expresses the semantic rules for the case in which the code needs to create a domain element from a constant value in the code (e.g. in an declaration associated with an initialization). Based on the status of the least significant bit, the code classifies the value as either an odd or even value in the domain.

```
public _IIntDomainInfo CreateFromConstant(int v, int srcLoc)
  {
      return(new RangeIntEvenOdd((v & 1 == 1) ? RangeIntEvenOdd.Odd : RangeIntEvenOdd.Even));
  }
public _IIntDomainInfo CreateUninitialized(int srcLoc)
  {
      return((_IIntDomainInfo) this.TopElt());
  }
```

Figure 113: Domain Element Creation from Uninitialized and Initialized Values.

10.1.2 Special Approximating Methods

This section examines several of methods whose implementation is required only of approximating domains. Neither of these implementations illustrates surprising functionality, but they do illustrate the rather lightweight requirements associated with serving as an approximating value domain[53].

10.1.2.1 Approximating Control Flow

Figure 114 shows the handling of the routines used by the analysis system to judge the truth status of predicates that must be calculated during abstract execution. Such routines need only be supported by the approximating semantics, as it is that domain which serves as the distinguished approximation to the current value. The conventions used for representing boolean values as integers designate non-zero values as "true", and zero values as "false". The even/odd domain offers only a crude approximation to values in general, and is too coarse to capture much information on the truth status of booleans. In particular, if a runtime value is established as *odd*, it is definitely true, but if a value is known to be *even*, it could either be true or false. These rules are encoded in the implementations of *FKnownTrue* and *FKnownFalse* seen in Figure 114.

```
public boolean FKnownTrue()
{
    if (this.m_posLattice == Odd)
        return(true);
    else
        return(false);
}
public boolean FKnownFalse()
{
    return(false);      // cannot KNOW that zero in either case
}
```

Figure 114: Routines to Assess Truth of Predicates During Abstract Control Flow

[53] This stands in contrast to the approximating *state* domain, which requires rather extensive support beyond that offered by other state domains.

10.1.2.2 Reporting Value Ranges

In addition to methods used to help dictate the course of control flow evolution, approximating value domains must also support a number of methods designed to allow external code to make use of information concerning the value approximation. An important need of the analysis code is for information on the precise value of runtime quantities when it is available. The two routines shown in Figure 115 are used for this purpose – the *FConcrete* method is used by the analysis machinery to find if the value being approximated is known to be true, and the *SSValue* method extracts precise information on a runtime value whenever it is available.

Even in those cases where the runtime value being approximated is imprecisely known, the analysis machinery or other domains may wish to make use of information on its associated range of possible runtime values. For example, the approximating state domain can make use of information on the interval in which a approximating value can fall in order to bound the possible sizes of allocations and to model regions of allocated areas known to be present for any possible allocation size. Routines *MaxPossibleSSValue* and *MinPossibleSSValue* (shown in Figure 116) are used for this purpose.

```
public int SSValue()
{

GlobalUtility.Assert(false, "Illegal invocation of routine that requires a concrete value on
a non-concrete.");
    return(0);
}
public boolean FConcrete() { return (false); }
```

Figure 115: Routines Used to Characterize the Approximation to a Runtime Value by the Approximating Value Domain.

```
public int MaxPossibleSSValue()
{
    switch (m_posLattice)
    {
    case Even:
        return(0x7fffffff - 1);
    case Odd:
        return(0x7fffffff);
    default:  …
    }
}
public int MinPossibleSSValue()
{
    switch (m_posLattice)
    {
    case Even:
        return(0x80000000);
    case Odd:
        return(0x80000000 + 1);
    default:  …
    }
}
```

Figure 116: Routines For Extracting Bounding Information for Partially Known Values.

10.1.3 Modeling Value Operators

Figure 117 shows the implementation of the semantic rules for integer addition within the even/odd semantics. The code (located in the class *RangeIntEvenOdd*) handles combinations between domain elements each of which is definitively known to be either odd or even. Other cases are delegated (by use of commutativity of addition) to handling by the top and bottom domain elements. For cases in which definitively even and odd elements are being combined, the rules as encoded in the method are familiar: If we have a situation in which operands are either both even or both odd, then the result of adding the two operands is even. Otherwise, it is odd. The implementation of many other binary integral operators represent minor variations on the schema seen in Figure 117.

```
public _IIntDomainInfo operatorAdd(_IIntDomainInfo argRHS)
  {
      if (argRHS instanceof RangeIntEvenOdd)
      {
          RangeIntEvenOdd argRHSNarrowed = (RangeIntEvenOdd) argRHS;

          if (this.m_posLattice == argRHSNarrowed.m_posLattice)
              // if equal types, result is even
              return(new RangeIntEvenOdd(Even));
          else
              // otherwise, result is odd
              return(new RangeIntEvenOdd(Odd));
      }
      else
          return(argRHS.operatorAdd((_IIntDomainInfo) this));
  }
```

Figure 117: The Implementation of the Semantic Rules for the Addition Operation in the Even/Odd Domain.

10.1.4 Handling the Least Upper Bound Operator

The final interface method to be examined by this class is the routine that takes the least upper bound of two domain elements. This routine is shown in Figure 118. The operation of the domain extends directly from the lattice shown in Figure 112. The routine is only invoked in cases where the left hand side of the operation is known to be either Even or Odd (this routine is drawn from the class *RangeIntEvenOdd*; other left hand sides are included in other classes). If both domain elements are at the same point in the lattice, that point is returned. Otherwise, if the right hand side is bottom, the left-hand side is returned. In all other cases handled by *FiniteHeightLUB*, the "top" value is returned.

```
  // curr <- LUB(this, curr).   fChanged = (LUB(this, curr) > this)
 public _IIntDomainInfo FiniteHeightLUB(_IIntDomainInfo argRHS, boolean[] fChangedOut)
 {
     if (argRHS instanceof RangeIntEvenOdd)
     {
       RangeIntEvenOdd argRHSNarrowed = (RangeIntEvenOdd) argRHS;

       // classification same => stays same
       if (argRHSNarrowed.m_posLattice == this.m_posLattice)
       {
           fChangedOut[0] = false;
           return(this);
       }
       else
       {
           fChangedOut[0] = true;
           return(new TopIntEvenOdd());
       }
     }
     else if (argRHS instanceof BottomIntEvenOdd)
     {
           fChangedOut[0] = false;
           return(this);
     }
     else
     {

     GlobalUtility.Assert(argRHS instanceof TopIntEvenOdd, "unrecognized class of IntEvenOdd in
 FiniteHeightLUB for RangeIntEvenOdd");
         fChangedOut[0] = true;
         return(new TopIntEvenOdd());
     }
 }
```

Figure 118: The Implementation of the Least Upper Bound Operator in the Even/Odd Domain.

10.2 Sign Approximating Domain

The second toy approximating value domain we will examine here is slightly less contrived than the Even/Odd domain, and has a long and distinguished history of use in the abstract interpretation literature. The sign domain approximates integer values into three categories according to their sign; a value can also be classified as $\top$ (completely unknown) and $\bot$. The lattice associated with this abstract domain is depicted in Figure 119. Note, importantly, that the domain does *not* have the capacity to describe values as *combinations* of signs (e.g. non-negative).

As might be expected, while the even/odd domain is of almost purely pedagogical interest, the sign domain could offer some limited insight into the functioning of numerically intensive code. As we will see below, however, the implementation of the domain follows the general patterns seen in the handling of the Even/Odd domain, and requires only slightly more work to implement.

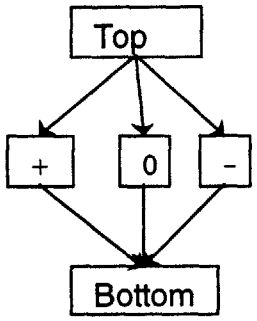

Figure 119: The Sign Approximating Value Domain Lattice.

10.2.1 Domain Element Creation

Figure 120 illustrates the implementations of the methods that create new domain elements from either uninitialized declarations or from a constant. The resemblance of the routines to those seen in Section 10.1.1 is no accident, and arises from the fact that all approximating value domains approximate a particular value. Any implementation of a approximating value domain is therefore logically required to associate a top lattice element with an uninitialized value, and must "classify" any initialization constant in accordance with the classifications implicit in its lattice.

```
public _IIntDomainInfo CreateFromConstant(int v, int srcLoc)
  {
      return(new RangeIntSign(((v < 0) ? -1 : ((v > 0) ? 1 : 0))));
  }
public _IIntDomainInfo CreateUninitialized(int srcLoc)
  {
      return((_IIntDomainInfo) this.TopElt());
  }
```

Figure 120: Sign Domain Implementations of Domain Element Initialization.

10.2.2 Modeling Value Operators

Figure 121 shows the code that handles the addition of two Sign Domain elements whose classification is precisely known. This method is implemented in the *RangeIntDomain* class; methods in other subclasses of *IntDomain* handle combination with the $\top$ and $\bot$ domain elements. As was the case for the Even/Odd domain, the rules behind the code shown are the commonsensical manipulation rules for two values for whom only the signs are known: If two elements share the same sign, the result of the add is the same sign. Otherwise, if either of the elements is known to be zero, the result is of the same sign as the other (non-zero) element. Otherwise, we cannot determine the sign of the result, and must return the top lattice element. As might be expected, many of the other binary operators can be implemented using a similar template to that

shown in Figure 121; in essence, the code merely serves as a compact coding device for the table of results that is in turn dictated by the rules of abstract interpretation.

```
public _IIntDomainInfo operatorAdd(_IIntDomainInfo argRHS)
  {
      if (argRHS instanceof RangeIntSign)
      {
          RangeIntSign argRHSNarrowed = (RangeIntSign) argRHS;
          if (this.m_posLattice == argRHSNarrowed.m_posLattice)
              return(this);
          else if (this.m_posLattice == SignZero)
              return(argRHS);
          else if (argRHSNarrowed.m_posLattice == SignZero)
              return(this);
          else
              // ok, must have opposite signs
              return(new TopIntSign());
      }
      else
          return(argRHS.operatorAdd((_IIntDomainInfo) this));
  }
```

Figure 121: Adding Domain Elements in the Sign Domain.

10.2.3 Handling the Least Upper Bound Operator

The final bit of code that we examine here implements the abstract rules underlying the least upper bound operator. Unlike the addition operator shown in Figure 121, this method is designed to handle values from all heights of the lattice: The $\top$ domain element (implemented in the subclass *TopIntSign*), the $\bot$ domain element (as implemented by *BottomIntSign*), and elements whose classification is known (implemented as instances of *RangeIntSign*). The rules for performing the least upper bound are just as would be expected from the lattice depicted in Figure 119. It requires only slightly more effort to set a boolean that indicates whether the result is higher in the lattice than the left-hand side argument ("this" in the method).

```
public _IIntDomainInfo FiniteHeightLUB(_IIntDomainInfo argRHS, boolean[] fChangedOut)
  {
      if (argRHS instanceof RangeIntSign)
      {
        RangeIntSign argRHSNarrowed = (RangeIntSign) argRHS;

        if (argRHSNarrowed.m_posLattice == this.m_posLattice)
        {
            fChangedOut[0] = false;
            return(this);
        }
        else
        {
            fChangedOut[0] = true;
            return(new TopIntSign());
        }
      }
      else if (argRHS instanceof BottomIntSign)
      {
            fChangedOut[0] = false;
            return(this);
      }
      else
```

```
        {

        GlobalUtility.Assert(argRHS instanceof TopIntSign, "unrecognized class of IntSign in Finit
eHeightLUB for RangeIntSign");
            fChangedOut[0] = true;
            return(new TopIntSign());
        }
    }
```

Figure 122: Handling of the Least Upper Bound Operator for the Sign Domain.

10.2.4 Conclusion

The sign domain is extremely simple to implement, but unlike the "Even/Odd" domain seen in Section 10.1, it is not limited to toy application. Relative to other representations of partially known values discussed in the next chapter, the sign domain offers a number of advantages. In particular, it allows for compact, three-bit value representation, high-efficiency value combination and implementation of the least upper bound operator, rapid convergence and a value classification system relevant for performing certain types of optimization.

10.3 Chapter Conclusion

This chapter has examined two simple integer value representations. Each of these representations is associated with a fixed lattice structure, and approximates values according to a set of disjoint categories. The semantic rules for these categories were uniformly simple, and arose in a straightforward manner from the mathematics underlying abstract execution. Having seen two examples of simple approximating domain implementations, we make use of TACHYON as a testbed for rapidly investigating value representations, and investigate two approximating domains that bypass traditional shortcomings and give the user the flexibility of choosing the precision with which the program should be analyzed.

Chapter 11 Example Approximating Value Domains, Part II – Extensible Domains

11.1 Introduction

This chapter examines two novel approximating domain representations that offer widely scalable precision. Both of these representations are associated with significant overhead, and primarily intended for research use into patterns of information available during analysis and precision tradeoffs. But the representations could also see active use in certain practical analysis contexts. The scalability and customizability of the domains is particularly attractive in the context of analysis extensibility, they allow a particular approximating value domain to be used in conjunction with a wide variety of collecting value domains, many of which could require different levels of analysis precision..

The abstractions presented in this chapter maintain their characteristic of scalability in quite different ways. One of the representations combines and generalizes the sign domain of Section 10.2 in order to capture information concerning the "buckets" in which a program value can fall within a user-defined sequence of intervals (buckets). This representation avoids some of the shortcomings associated with interval representation while retaining many of its advantages and allowing for scalable precision. By contrast, the other representation takes a novel approach to scalability and value representation, based on characteristics of the analysis process discussed in depth in Appendix 1. While distinctly unconventional, this "sample space" approach offers some compelling advantages over event space representations for situations for those situations in which the heavy analysis overhead is acceptable.

The chapter commences with a "wish list" of desirable characteristics for representations. With these characteristics in mind, we then turn to a discussion of each of the extensible representations. Finally, the chapter concludes with a higher-level discussion of the representations.

This chapter should ideally be read in conjunction with Appendix 1 and Appendix 2, both of which provide background information and discussion that will be very helpful in understanding this chapter. Appendix 2 investigates a number of existing value representations that have been used in the literature, compares their strengths and weaknesses in light of the "wish list" presented in Section 11.2. That appendix helps to highlight some areas in which existing representations exhibit shortcomings, and helps to motivate the presentation of the extensible representations. Appendix 1 presents a probabilistic model of program execution that forms the foundation and inspiration for the "sample space domain" representation introduced in Section 11.4. Study of the appendix will make that section considerably more comprehensible.

11.2 Representational Goals

In order to conduct high-quality analysis, it is desirable to transcend the known/unknown dichotomy, and to allow for the representation of *partially known* values. There are number of desirable qualities to which we would like to give weight in such a representation. Some of the most important of these qualities are listed below:

- **Representational Breadth**: A representation should ideally be capable of accurately representing many varieties program values of – for instance, values drawn from different data types. More importantly, it would be particularly attractive to have a representation that could represent both the entire domain of possible concrete values taken on by a program value, as well as a sparse subset of that domain. For example, while the even/odd domain can represent the entire domain of possible values, it is relatively crude in its "steps" of approximation – any knowledge concerning a program value is diluted to the same level.

- **Compact Representation.** Program values are the quanta of which a program state is composed, and an accurate model of program state typically entails the use of a very large number of program values. To minimize the memory (and, indirectly but important, the time) required by analysis, it is important that values be represented in as compact a way as possible.

- **Efficiency of Analysis Manipulation.** The use of a value representation and its associated abstract domain should ideally impose low performance cost during analysis. Our attention here is focussed on two types of efficiency:

  - **Cost of Value Operations.** Each time an operator is applied to a value, a certain amount of time is required to compute the resulting value. Certain abstract domains admit to faster computation and manipulation than do others: For example, combining two values during constant propagation (such as that illustrated in [Chen 1994]) is a constant-time operation, regardless of the particular elements from the abstract domain being combined. By contrast, the time required to combine values drawn from a "set representation" abstract domain (such as that used in [Osgood 1993] and [Weise, Conybeare et al. 1991]) can be far more variable: In general, the further up the domain a value is located, the more expensive will be its combination. If we are given two values drawn from this domain of cardinality m and n, the time required to compute the resulting value is $o(mn)$. Because the analysis process involves modeling a myriad of such value combinations, the use of such a domain can impose a substantial computational cost on analysis. [Osgood 1993]

- **Height of the Abstract Domain.** As was discussed in Chapter 2, the program analysis process can be viewed at an abstract level as the calculation of a fixed point of a collecting interpreter $I_{C,P}^{\#}$. A key factor in determining how long this process will take to complete is the length of the longest chain in the abstract state domain State$^{\#}$ [54]). Intuitively, this bounds the number of times we may have to take the least upper bound of two states in a program loop until we reach a state (located at the top of the chain) that serves as a legitimate approximation to *any* of the states encountered during the loop. The height of the abstract state domain is closely linked in turn to the height of the domain from which the values within the state are drawn. For example, if we have a state consisting of n variables, each of which are associated with values drawn from a value domain of height h, the (theoretical) height of the State$^{\#}$ domain is $O(nh)$. Thus, the greater the height of the abstract domain, the longer the amount of time an analysis can take in theory, and frequently will take in practice.

These costs do not always go together. While a set-based abstraction is expensive both in per-operation cost and in the height of the domain, a "range" representation of allows for rapid per-operation processing, but impose significant costs by increasing the height of the abstract state domain.

- **Closure Under Operations.** High-quality modeling the behavior of program values during analysis requires the effective modeling of operations performed on those values. A fundamental requirement for effective modeling of an operation is the ability for a domain to adequately represent the outcome of such operations. Some popular value representations do not have *accurate* closure over the set of operations through which they can be combined. In other words, it is possible to that the result of combining well-characterized values represented in the specified manner will yield a result that cannot be expressed in this way except by the assumption of the $\top$ value in the lattice. [Osgood 1993] For example, applying the modulo or division operator to a range representation yields a result not conducive to description as an interval. Again with the goal of providing maximal user choice over the operation of program analysis, we would like to be able to offer the user the option of a high quality representation of the result of as many operations as possible.

- **Scalable Analysis Precision.** A dominant theme within this thesis is the desire to permit user access to and control over as much of the analysis process as possible. An important component of this process is providing the user with fine-grained mediation of the tradeoff between the analysis precision and running time for all or portions of a program. Analysis quality and computational demands are both

[54] We also refer to this as the "height of the domain State$^{\#}$"

strongly affected by the character of a value domain, and allowing precise control over the computational demands exacted with that domain is an important ingredient in throttling the analysis process as a whole. In addition to the desire to provide the typical user the option of performing high-precision analysis on a segment or all of a program, there is another motivation for permitting the user of a highly precise value representation: Such a representation would be a boon to research at two levels:

- Research into patterns in programs. This study could help recognize opportunities for optimization and improved understanding by identifying yet untapped patterns in program execution and the ways in which uncertainty in aspects of program execution context induces uncertainty throughout the program. Such research would make use of the more powerful representation as a precise and powerful tool for studying the characteristics of real-life programs.

- Research into alternative collecting domains. This is a "meta-level" study whose goal would be to leverage the powerful (but potentially inefficient) representation to identify those components of value representation and analysis algorithms that are most valuable for program analysis (either in general or when targeted for particular applications). In this case, the more powerful representation is being used as a research vehicle for deriving alternative representations that would be better suited to certain analysis needs.

- **Sensitivity to Relative Frequencies of Different Program Values.** Past value models such as those based on interval or set-based representations were capable of indicating the possible concrete values taken on by a program value, but did not maintain any information capable of expressing the relative frequencies attached to different possible values. Such information can play an important role in understanding system behavior, plays a critical role in forming the patterns discussed in Appendix 1. Assumptions to the effect that all possible values are equally likely are inconsistent, in that combinations of uniform values naturally induce values with distinctly non-uniform value distributions. Appendix 1 contains extensive discussion related to these issues.

- **Capacity to Model Sample Space Information.** Appendix 1 discusses the modeling of program values as random variables, each associated with a sample space of possible configurations and a derived (standard) event space. Because the event space abstracts away all information concerning the contexts in which different possible values apply, it is not possible to maintain inter-value *dependency information* using event space information alone. Appendix 1 and Appendix 2 also noted that the result of combining program values can depend critically on contextual information. Indeed, combination of

pairs of values combinations which appear identical from the perspective of event space (e.g. with corresponding elements of each pair known to be in the same interval, set, or to satisfy the same predicates) can yield values differing strongly in both sample and event space. As discussed in Appendix 1, some of the most prominent patterns seen in program execution arise from dependency between program values, and in many cases these dependencies represent significant barriers to traditional system analysis. [Taylor 1995] Effective modeling of dependency information is a highly desirable in any analysis that aspires to provide the option of modeling of program behavior at very high precision, for the benefits of modeling values with high precision are quickly lost without knowledge of the dependencies between such values.

Having enumerated some of the more important goals for a representation of partially known values, we turn now to a discussion of the first of the two extensible representations we consider in this chapter.

11.3 Disjoint Interval Approximating Value Domain

The sign domain of presented in the previous chapter offered a number of advantages relative to other partially known value representations. In particular, it allowed for compact value representation, high-efficiency value combination and implementation of the least upper bound operator, rapid convergence born of shallow domain height, and a value classification system relevant for performing certain types of optimization. Unfortunately, the domain presented is also of fixed structure, and offers neither a scalable nor a customizable basis set of primitives for performing analysis. This section examines a generalization of this approach that allows for a classification of run-time values into different possible user-defined intervals. The user is free to select as many intervals as desired, and free to define the endpoints of those intervals in accordance with the type of information desired and constraints on the running time of the analysis.

11.3.1 Structure of the Approximating Value Domain

Within the sign domain of Section 10.2, only five domain positions were possible: $\perp$, -, 0, +, and $\top$ (see Figure 123). The domain presented in this section generalizes this domain in two ways:

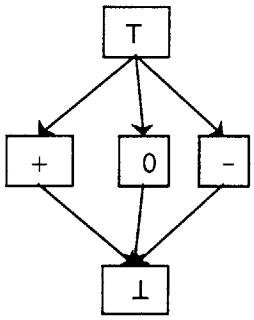

Figure 123: The Topology of the Sign Abstract Domain Introduced in Chapter 10.

- **Custom Partition Definitions.** Within the disjoint interval domain, the user is allowed to specify the ordered sequence of partitions used to classify values. For example, rather than classifying values into negative, zero, and positive categories, the user could establish additional categories exactly representing positive and negative one (a change allowing for some additional strength reductions in code optimization). Another domain (reminiscent of the approach of [Ayers 1993]) might additionally break out manifest constants expressed in the program text, but aggregate all other constants together. The definition of custom partition structures allows for the user to shape the domain so as to express information regarding those characteristics of values that are of interest to the particular collecting domains being run, while avoiding wasting the computational resources needed to represent unnecessary characteristics of domain structure.

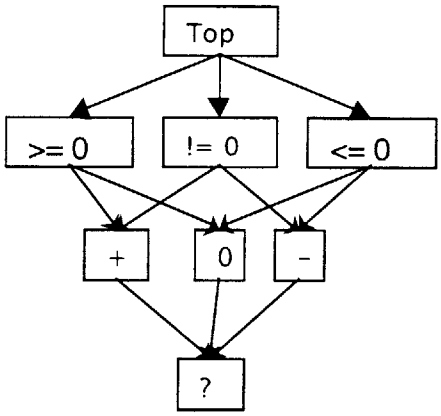

Figure 124: A Disjoint Interval Sign-Classification Domain.

Powerset Semantics. In addition to classifying a value as definitively belonging to a single partition of the value domain, the current domain can express the fact that an abstract value is known to lie in any *set* of such partitions. Thus, while the sign domain of Section 10.2 would be forced to approximate a value that is known to be non-negative as the ⊤ element, the disjoint interval sign implementation can be more accurate. In particular, the disjoint interval domain would express such a value as belonging to *either* the negative partition or the zero partition. (See Figure 124). The powerset domain that results is reminiscent of the set domain discussed in Section 2.3, with an important difference: Because values are "pigeonholed" into a relatively compact set of bins, the powerset in the partition domain has a vastly lower natural[55] height than that associated with the set domain. (For the powerset domain, the natural height of the domain is equal to the number of partitions. In the set domain – in which each possible integral value is essentially a partition – the natural domain height is equal to the number of distinct integral values.)

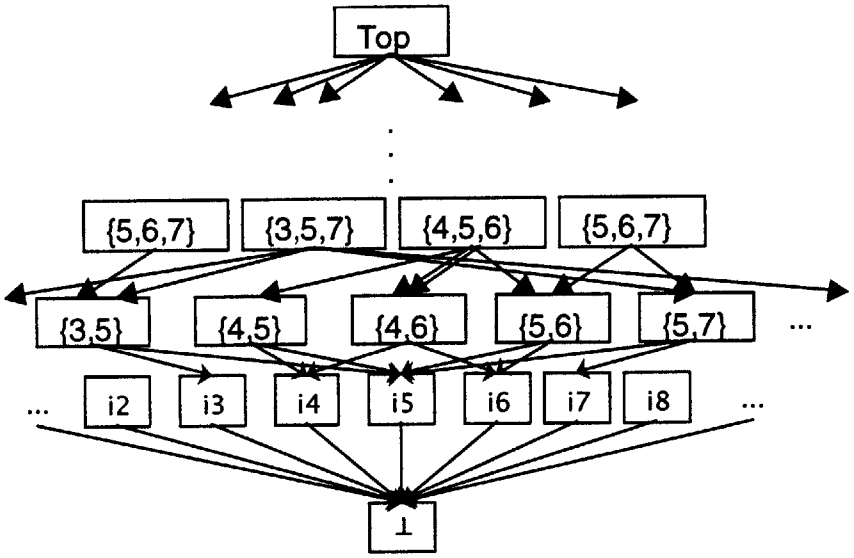

Figure 125: The General Structure of Disjoint Interval Domain Lattice.

A visual expression of the resulting abstract domain structure is shown in Figure 124 and Figure 125. Figure 124 provides an illustration of the domain structure for a particular set of partitions, while Figure 125 depicts the domain structure for no particular partition structure, and is thus somewhat abstract. The first row in Figure 125 above the bottom element corresponds to sets of concrete values known to belong to

[55] The word "natural" here is used to mean the height of the powerset domain in the absence of the ad-hoc limit imposed by a fixed height requirement. See Section 2.3 for more detail.

precisely one user-defined interval (e.g. *i2*). Higher levels in the lattice of height *n* correspond to sets of concrete values that are known to fall into precisely *n* different intervals.

11.3.2 A Note on Representation

The last section briefly presented the structure of the abstract domain. For simplicity, compactness, and efficiency of combination, the *simultaneous* use of approximating domain values supporting different partitionings are not currently supported. The use of a partitioning lattice allows values to be represented as bit vectors. Each bit within the vector corresponds to a particular partition; if the bit is set, it indicates that it the run-time values approximated by the current abstract domain element may fall in the range of integral values associated with the partition. A bit which is not set indicates that the run-time values could definitely *not* hold *any* values in that range.

The distinguished $\top$ value corresponds to a bit vector with all bits set (indicating that the run-time values could hold any value), while $\bot$ can be represented as a bitvector with *no* bits set. The use of only a single partitioning guarantees that the interpretation of each bit position is the same across elements drawn from the same domain.

Within the current implementation, the user specifies the partitioning of the abstract domain by specification of an array consisting of the minimum point in each partition. Because the establishment of only a single partitioning is supported, the abstract domain structure induced in this manner remains in effect throughout the analysis.

11.3.3 Domain Element Creation

As illustrated in Figure 126, the mechanisms needed to create new elements are minimal. As required for a approximating domain representation, uninitialized values are associated with the top element of the domain. Manifest constants are simply placed into the appropriate partition based on the value of the constant – a generalization of the behavior seen in the Sign Domain of last chapter.

```
public _IIntDomainInfo CreateFromConstant(int v, int srcLoc)
{
    return(s_MRGuidingValueIntReturnValue = FactoryMethod(this.BitSetFromExactValue(v)));
}
public _IIntDomainInfo CreateUninitialized(int srcLoc)
{
    return(s_MRGuidingValueIntReturnValue = FactoryMethod(this.BitSetForEntireRange()));
}
protected BitSet BitSetFromExactValue(int i)
{
    int ctCategories = this.CtCategories();
    // ok, lowre boudn could be anywhere
    int iCategory = this.LocateContainingCategoryInCategoryRange(i, 0, ctCategories - 1);
    // ok, now mark all bits between iCategory and jCategory
    BitSet bs = new BitSet(ctCategories);

    bs.set(iCategory);
    return(bs);
}
```

Figure 126: Routines to Create Domain Elements from Declarations and Manifest Constants

11.3.4 Modeling Value Operators

The combination of program values is the most complicated and computationally expensive aspect of analysis using the disjoint interval value domain. Each of these shortcomings arises from different characteristics of the domain. The following two subsections briefly examine each of these issues.

11.3.4.1 Computational Complexity

The high computational cost of the combination process stems from the need to apply semantic rules combining elements from all possible combinations of domain partitions. Recall that in general, two elements of the disjoint interval value domain that are combined can include possible values in any (non-empty) subset of the set of user-defined partitions. The value combination algorithm (implemented in the method *CombineWithBinaryOperator* shown in Figure 127) decomposes this problem into a series of pairwise combinations each involving only two partitions – one partition in which the left hand operand can lie, and another partition in which the right hand operand can lie. This decomposition requires a double-nested loop that takes time quadratic in the number of possible domains associated with each value (i.e. in the number of set bits in the bitvectors). As with the "set" representation discussed in Appendix 2, the quadratic expense of this computation directly reflects the inablity of the system to identify the *dependencies* between the left hand and right hand operands, and thus to recognize which potential run-time values are simultaneously possible and which are not. In the absence of this information, the results of *all* possible combinations must be approximated.

Despite its high cost, the code for *CombineWithBinaryOperator* is quite simple: Each pair of partitions in which the run-time value can lie is combined using the semantic rules for partition combination under the appropriate operator. (e.g. semantic rules for partition combination using the *add* operator, or *mul* for

multiplication) Each such combination yields a set of output partitions in which the resulting value can lie (represented as a *BitSet*). The full set of possible results is accumulated with the union operator. The code then returns the accumulated set, which approximates the set of all possible partitions that could result from a combination of any possible run-time value in the left hand operand with any possible run-time value in the right hand operand.

```
    public _IIntDomainInfo CombineWithBinaryOperator(IntDisjointOrderedCategoriesDomainBinaryOpe
ration op, _IIntDomainInfo argRHS)
    {
        DisjointOrderedCategories argRHSNarrowed = (DisjointOrderedCategories) argRHS;

        // basically, we superimpose all combinations of possible combinations

        if (this.FIsBottomElt() || argRHSNarrowed.FIsBottomElt())
            return(s_MRGuidingValueIntReturnValue = (_IIntDomainInfo) this.TopElt());
        else
        {
            BitSet  acc = null;

            for (int i = 0; i < this.CtCategories(); i++)
                if (this.m_categories.get(i))
                for (int j = 0; j < argRHSNarrowed.CtCategories(); j++)
                    if (argRHSNarrowed.m_categories.get(j))
                       {
                        BitSet vResult = op.CombineSingleCategoryPair(this, i, j);

                        if (acc == null)
                            acc = vResult;
                        else
                            acc.or(vResult);
                       }

            return(s_MRGuidingValueIntReturnValue = FactoryMethod(acc));
        }
    }
```

Figure 127: Code to Perform the Generic Binary Combination in the Disjoint Interval Approximating Domain.

11.3.4.2 Software Engineering Complexity

In contrast to the computational complexity of combination that stems primarily from the combinatorial character of value combination, the *software engineering* complexity arises primarily from the need to describe the semantics of interval combinations for different operators. While the computational complex is largely fixed across different operations (as evidenced by the structure of Figure 127, the code to combine specified pairs of partitions is far more complicated for certain interval combination operations than for others. For example, Figure 128 shows the code designed to handle the combination of two partitions for the addition operation. The method *CombineSingleCategoryPair* in the *AddIntDisjointOrderedCategoriesDomainBinaryOperation* class encodes the rules by which two partitions (identified by indices *i* and *j* passed in as arguments) calculate a set of possible result partitions. In the case of the addition operation, the mapping to output partitions from the pair of partitions being combined is particularly simple, due to the fact that the result of adding two intervals is its an interval.

By contrast, the code to handle a *division* operation is much more complicated (as suggested by the size and complexity of Figure 129). This additional complexity reflects the fact (discussed in Appendix 2) that the result of dividing a value lying in one interval by a value lying in another cannot always be expressed by an interval (and, even when it can, the endpoints of the resulting interval cannot be trivially calculated from the endpoints for the input intervals). In general, there is a high variance in the size and complexity of the code required to handle different operations. Moreover, the results of many of those operations (such as division) are likely to be sufficiently broad in practice to make detailed accumulations of the possible results of dubious value. A simple heuristic likely to significantly boost efficiency the code shown in Figure 127 would be to maintain information during operation of *CombineWithBinaryOperator* to determine if the partially accumulated value to be returned has already reached the top of the lattice. If so, no further combination of partitions need take place.

```
    public _IIntDomainInfo operatorAdd(_IIntDomainInfo argRHS)  {   return this.CombineWithBinar
yOperator(s_AddBinaryOperator, argRHS);  }
    private IntDisjointOrderedCategoriesDomainBinaryOperation s_AddBinaryOperator = new AddIntDi
sjointOrderedCategoriesDomainBinaryOperation();
    private static class AddIntDisjointOrderedCategoriesDomainBinaryOperation extends IntDisjoin
tOrderedCategoriesDomainBinaryOperation
    {
        public  BitSet CombineSingleCategoryPair(DisjointOrderedCategories context, int i, int j)
        {

    return(context.BitSetFromPossibleRange(context.MpICategoryMinValue(i) + context.MpICategoryM
inValue(j),context.MpICategoryMaxValue(i) + context.MpICategoryMaxValue(j)));
        }
    }
```

Figure 128: Handling Partition Combination for the Addition Operation.

```
private class DivIntDisjointOrderedCategoriesDomainBinaryOperation extends IntDisjointOrderedCat
egoriesDomainBinaryOperation
    {
        public  BitSet CombineSingleCategoryPair(DisjointOrderedCategories context, int i, int j)
        {
            // this is also tricky

    // we need to watch out for the case where the range by which we are dividing contains zero
            //  (in this case, the results lie OUTSIDE a certain range around zero)

            int negInfinity = 0x80000000;
            int posInfinity = 0x7fffffff;
            int a = context.MpICategoryMinValue(i);
            int b = context.MpICategoryMaxValue(i);
            int c = context.MpICategoryMinValue(j);
            int d = context.MpICategoryMaxValue(j);

            if (c <= 0 && d >= 0)
            {
                if (a == 0 && b == 0)
                return context.BitSetFromPossibleRange(0, 0);
                // handle various cases separately so don't get arithmetic errors dividing by 0
                if (c == 0 && d == 0)
                {
                if  (a >= 0 && b >= 0)
                    return context.BitSetFromPossibleRange(0, posInfinity);
                else if  (a <= 0 && b <= 0)
                    return context.BitSetFromPossibleRange(negInfinity, 0);
```

```
                else
                    //  (a <= 0 && b >= 0)
                    return context.BitSetFromPossibleRange(negInfinity, posInfinity);
                }
                else if (c == 0)
                {
                // ok, lower endpoint is at zero, other is positive

                if  (a >= 0 && b >= 0)
                    return context.BitSetFromPossibleRange(a / d, posInfinity);
                else if  (a <= 0 && b <= 0)
                    return context.BitSetFromPossibleRange(negInfinity, b / d);
                else
                    //  (a <= 0 && b >= 0)
                    return context.BitSetFromPossibleRange(negInfinity, posInfinity);
                }
                else if (d == 0)
                {
                // ok, upper endpoint is at zero, other is negative

                if  (a >= 0 && b >= 0)
                    return context.BitSetFromPossibleRange(negInfinity, a / c);
                else if  (a <= 0 && b <= 0)
                    return context.BitSetFromPossibleRange(posInfinity, b / c);
                else
                    //  (a <= 0 && b >= 0)
                    return context.BitSetFromPossibleRange(negInfinity, posInfinity);
                }

                if (a >= 0 && b >= 0)
                {
                // ok, here the intervals are from
                  //    -infinity to a/c
                  //    a/d to infinity

                BitSet bsIntervals = context.BitSetFromPossibleRange(0x80000000, a / c);
                bsIntervals.or(context.BitSetFromPossibleRange(a /d, 0x7fffffff));

                return(bsIntervals);
                }
                else if (a <= 0 && b <= 0)
                {
                // ok, here the intervals are from
                  //    -infinity to b/d
                  //    b/c to infinity

                BitSet bsIntervals = context.BitSetFromPossibleRange(0x80000000,b/d);
                bsIntervals.or(context.BitSetFromPossibleRange(b/c, 0x7fffffff));

                return(bsIntervals);

                }
                else
                {

    GlobalUtility.Assert(a <= 0 && b >= 0, "unexpected case in DivIntDisjointOrderedCategoriesDo
mainBinaryOperation");
                // if BOTH intervals span the range, any value is possible
                return(context.BitSetForEntireRange());
                }
            }
            // ok, if the denominator does NOT cross the axis, things are a lot easier
            else
            {
                // for simplicity and to reduce the risk of errors, we just find the max and min
                //of all possible endpoints

                int e1 = (a / c);
                int e2 = (a / d);
                int e3 = (b / c);
                int e4 = (b / d);
```

```
    return(context.BitSetFromPossibleRange(GlobalUtility.Min4(e1, e2, e3, e4), GlobalUtility.Max
4(e1, e2, e3, e4)));
            }
        }
    }
```

Figure 129: Sophisticated Reasoning for Combining Two Partitions For a Division Operation in the Disjoint Interval Domain.

11.3.4.3 Conclusion

While the software engineering complexity needed to approximate different operations on elements of the partition domain is worthy of note in the case of a few operations, such overhead applies equally to all domain customizations and does not scale with the size of the domain. As such, it imposes no important constraints on domain implementations. By contrast, the quadratic computational complexity associated with element combination does impose serious constraints on the scalability and widespread use of the domain. Given the tendency of interval approximations to lose precision as the number of operations through which they have been calculated rises, it seems questionable whether larger domain sizes will be sufficiently accurate (and thus valuable) to be worth their significant cost in combination. By leaving the number and boundaries of the partitions up to user control, the user can choose the appropriate balance between such tradeoffs.

11.3.5 Handling the Least Upper Bound Operator

In contrast to the value operations discussed above, the handling of the least upper bound operator of the partitioned value domain is very simple. Producing an approximation to both of the two values represented by bitvectors requires nothing more than taking the bitwise-"or" of the input bitvectors. The code to perform this operation is shown in Figure 130. Note that the implementation of the *FiniteHeightLUB* requires slightly more mechanism, due to the need to set the boolean indicating whether the result of the operation is higher in the lattice than the left-hand operand.

```
public _IIntDomainInfo LUB(_IIntDomainInfo argRHS)
{
    DisjointOrderedCategories argRHSNarrowed = (DisjointOrderedCategories) argRHS;

    BitSet bsReturn = (BitSet) this.m_categories.clone();
    bsReturn.or(argRHSNarrowed.m_categories);
    return(s_MRGuidingValueIntReturnValue = FactoryMethod(bsReturn));
}
```

Figure 130: The Least Upper Bound Operator for the Disjoint Interval Value Domain.

11.3.6 Conclusion

The disjoint interval value domain is a pragmatically designed event-space representation that carries many of the benefits of more compact representations while allowing for arbitrary precision scalability. Moreover, the abstraction allows the user to select a domain structure that is either compact or expressive those regions of the concrete domain the user judges most appropriate for the collecting domains being manipulated. (For example, a collecting domain that performs simple constant propagation is naturally paired with intervals representing only the manifest constants in the code. On the other hand, a collecting domain that recognizes and exploits opportunities for strength reduction might be fruitfully associated with domains of single length around the origin.)

The disjoint interval domain does, however, have two notable disadvantages. First, the per-operation cost for applying a binary operator scales quadratically with the number of partitions in the domain, making infeasible domains using large numbers of partitions. In addition, the results of applying operators to domain elements are not always as precise as would be desired. Both of these shortcomings stem primarily from the inability to recognize *dependency* information.

11.4 The Sample Space Representation

11.4.1 Fundamentals

11.4.1.1 Introduction

This section presents the "Sample Space Representation", a scalable model of program values that takes advantage of the observation that both the poor scaling and the limited precision of the disjoint value representation arise from a single root: The failure to model *dependencies* among values.

Appendix 1 examined program behavior in a probabilistic light. That discussion drew particular attention to the manner in which the probability distributions for program inputs induce uncertainty in program values throughout the execution of a program. To this end, the chapter demonstrated how program inputs and internal program values can be viewed as random variables defined over the sample space of possible execution contexts. This section builds on that understanding and on the discussion of abstract interpretation in Chapter 2 to construct a *software model* of the probability distributions associated with program values in the presence of uncertainty concerning program execution context. This value model can then be used as a approximating value domain during program analysis. Our goal here is to be able to conduct accurate and safe program analysis on programs with partially specified input and partially known values. By basing our model of uncertainty in program values on abstracted characteristics of program operation, we can construct a approximating domain that is general, powerful, and precise. As a bonus, this model of uncertainty in program values allows for more precise analysis of control flow constructs as well.

In a nutshell, the sample space representation is a compact means of carrying around a (potentially sparse and probabilistic) representation of a program variable's values mapped over a user-defined sample space corresponding to all possible execution contexts. When values are combined, combination always takes place only between values (or "samples") associated with identical positions in sample space. If keeping the full set of program values is judged to be too expensive, the information on some of those samples may be discarded at the user's discretion. Dealing with the potential absence of certain samples within the abstraction require some additional complexity in value manipulation, but add to its general usability by allowing for lighter-weight application in those contexts where probabilistic analysis is acceptable.

11.4.1.2 Event Space and Sample Space Representations

11.4.1.2.1 Introduction

Section 11.2 characterized some of the qualities desirable in a value representation. While this "wish list" is helpful for delineating the properties for which we'd like to optimize in a representation, it is only half the story: It provides no information on the constraints that obtain in designing a representation. This section discusses one of the most fundamental – but least discussed – choices that obtains in the selection of a representation. Motivated by the tradeoffs discussed in this discussion, we take a very different approach to value representation than is seen in other analysis systems. The next section then begins to fill in some of the details of this approach.

11.4.1.2.2 Values as Random Variables

Appendix 1 explored the nature of uncertainty in program analysis. One outcome of this exploration was the recognition that the value of a program quantity at a particular point during analysis could be completely characterized as a random variable associated with particular values for each point in a sample space of possible execution contexts. Taken in the context of a discussion of value representation strategies, this is a heartening result, as it places a strict upper limit on the amount of information needed to represent a value with maximal precision.

11.4.1.2.3 Limitations of Naïve Sample Space Representations

Given the conceptual model noted above, it seems reasonable to aspire to using the sample space characterization of program values as the basis for an accurate value representation, and specifically for a representation of *partially known* values. Unfortunately, this is not always straightforward to do: While the set of possible execution contexts that defines the sample space can be quite compact, it can also be tremendously large. This is particularly true in the contexts of multidimensional sample spaces, which suffer from the "curse of dimensionality" – the fact that the size of the space rises geometrically with the number of dimensions. (For example, a circuit that holds 10 slots, each of which can accommodate 5

different components would have a sample space of size 5^{10} – a truly massive number of possible execution contexts.) In cases where program inputs occur dynamically throughout the operation of the program, the sample space can be of unbounded size, and the explicit representation of that sample space is logically impossible.

11.4.1.2.4 Representation Options

Faced with the infeasibility of representing the entire sample space associated with a program values, there are three fundamental options that obtain when designing a value representation:

- The goal of representing partially known values can be abandoned altogether.
- Partially known values can be characterized by a relatively small number of values from value event space.
- Partially known values that can be characterized by a relatively small number of values from value sample space.

The large majority of existing program analysis systems take the first option, choosing to sacrifice the ability to represent partial knowledge concerning program values for the sake of computational frugality. In many cases this is indeed the most suitable strategy. But as noted in Section 11.2, there are many cases in which it would be desirable to provide the user with the option of maintaining a stronger form of representation. In such cases, we need to find a way to summarize the full sample space of the value being represented in a much more compact manner. There are two domains from which this summary data could be drawn: From the value's event space, or from the value's sample space.

11.4.1.3 Event Space Representations

Virtually all[56] previous program analysis systems that are capable of maintaining information on partially known program values represent those values using information drawn purely from the *event* space of the value. In other words, the variation of a program value over the sample space of possible execution contexts is summarized by information drawn from the probability distribution of *concrete values* that can be taken on by the program value. By providing insights into salient characteristics of the distribution of possible values taken on by a program quantity, event-space statistics can be extremely useful for the purposes of describing values to the user.

---

[56] The sole exception of which the author is aware is the system dynamics analysis package described in [Eberlein], which is capable only of representing values that share identical linear sample spaces across all values within the simulation.

One of the most important advantages of maintaining event space information lies in the capacity to extract compact information from the event space so as to make statements that are guaranteed to be accurate over the entire set of possible execution contexts. Event space information that used in describing program values is typically summarized from global features of the *entire* event space (that is, it is sensitive to characteristics of the random variable being summarized over all regions of its sample space). Such features are selected so as to conservatively bound or describe the possible values associated with the random variable over the entire sample space. To cite two examples of representations discussed in Appendix 2, the set representation maintains a set containing all of the possible values taken on by the random variable, while the two boundaries of the interval representation are guaranteed to compactly demarcate the span of that random variable.

Because the information typically used to represent a value conservatively summarizes the properties of the entire event space of that value, algorithms can leverage it in a strong manner. For example, if it is known that the value of a particular program quantity falls in a particular interval of values, any predicate that tests the equality of that quantity to a value lying outside the range will be known to be false.

While the use of event space information to represent a value does have some important advantages, it is inevitable that any representation that discards information regarding the probability distribution of a value will suffer shortcomings during analysis. Event-space representations are no exceptions, with the most significant of these difficulties is the poor precision in value combination that results from the lack of inter-value contextual information.

As discussed in Appendix 1, the sample space of a program value disaggregates all conceptually distinguishable execution contexts. The event space for the variable aggregates together all execution contexts that induce identical concrete values on that program value, regardless of the execution context of origin. As a result, maintaining a representation that preserves only event space information regarding program values discards entirely the contextual information associated with those values – and thus any information regarding the *dependencies* between values.

The key difficulty is that program values being combined may be associated with event spaces of very different character. In particular, the partitioning of the underlying sample space into contours which the random variable assigns to equal values may be entirely different between each of the two values. Suppose we are computing $v_1 \oplus v_2$ using two values v_1 and v_2 that are represented only with event space information.

Because of the aggregation inherent in the shift to an event space, there is no way of knowing from the event space information which possible values of v_1 and v_2 can actually occur in the same context. Put more informally yet, when a binary operator combines v_1 and v_2 there is no way of knowing "which possible value from v_1 goes with which possible value from v_2." As a result, even in the presence of complete information describing the *event* space of the two values being combined, it is impossible to precisely compute the event space characteristics of the value resulting from such a combination. Figure 133 and Figure 134 graphically illustrates these limitations by depicting the event space that actually results from the application of the same binary operator to two different pairs of input values. Despite the fact that the pairs of inputs share identical event spaces, the event spaces resulting from the combinations differ dramatically.

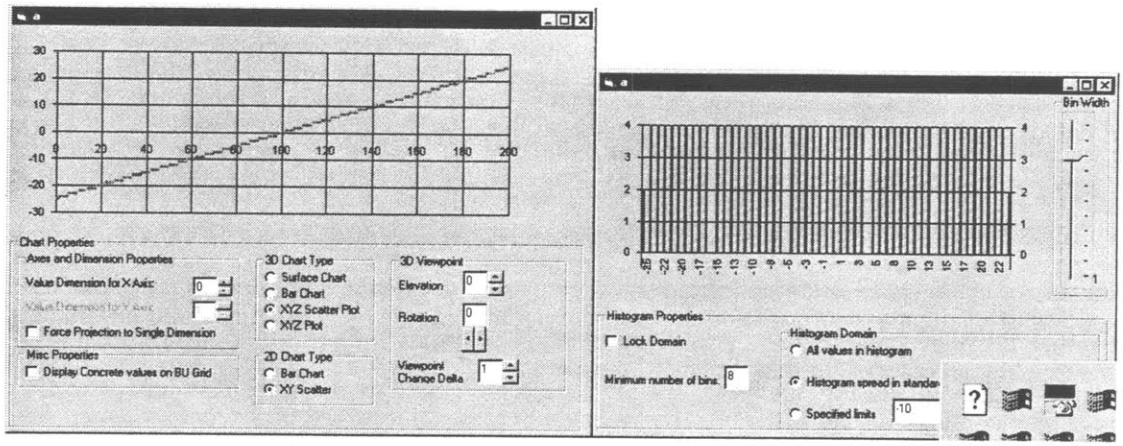

Figure 131: Event and Sample Space of *a*

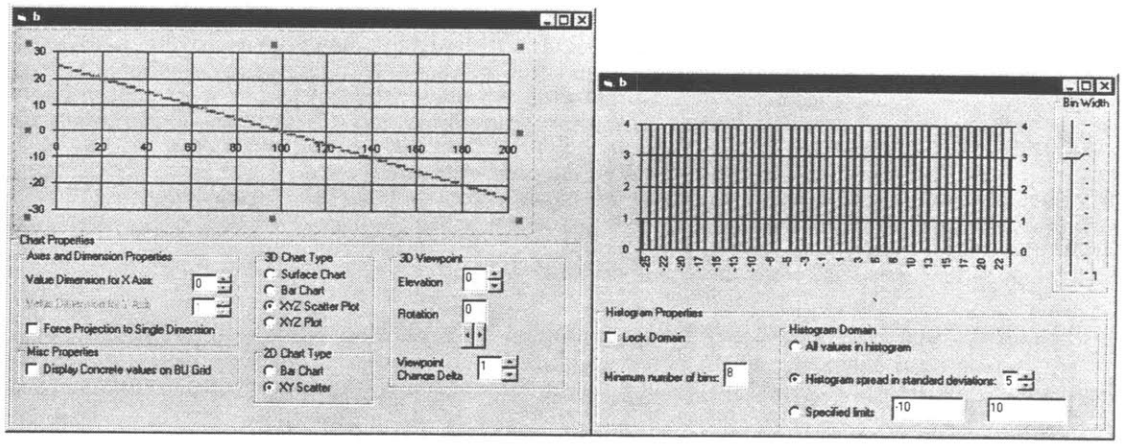

Figure 132: Event and Sample Space of *b*

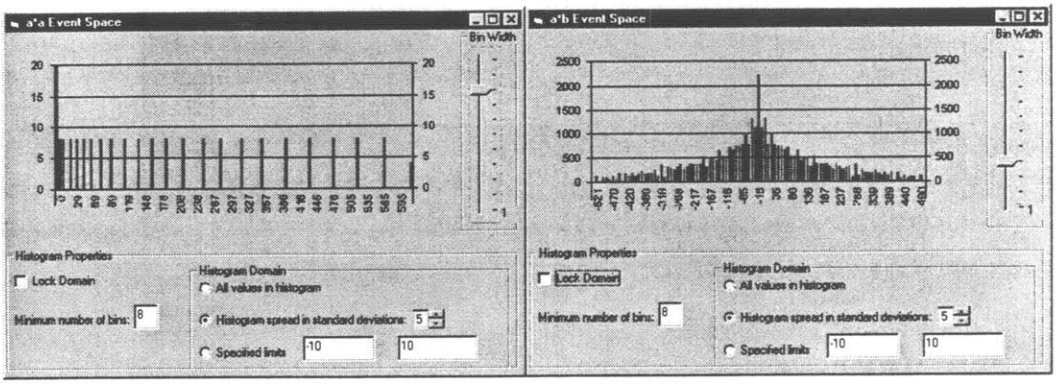

Figure 133: Comparison of Event Space of $a*b$ and $a*a$

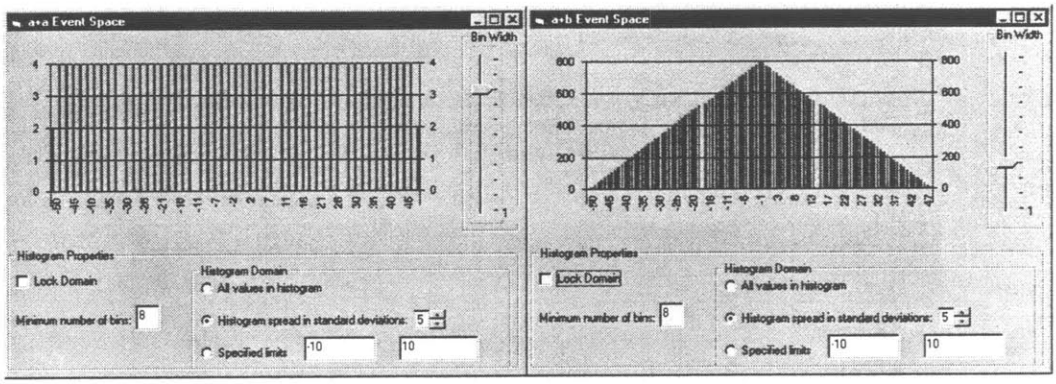

Figure 134: Comparison of Event Space of $a+a$ and $a+b$

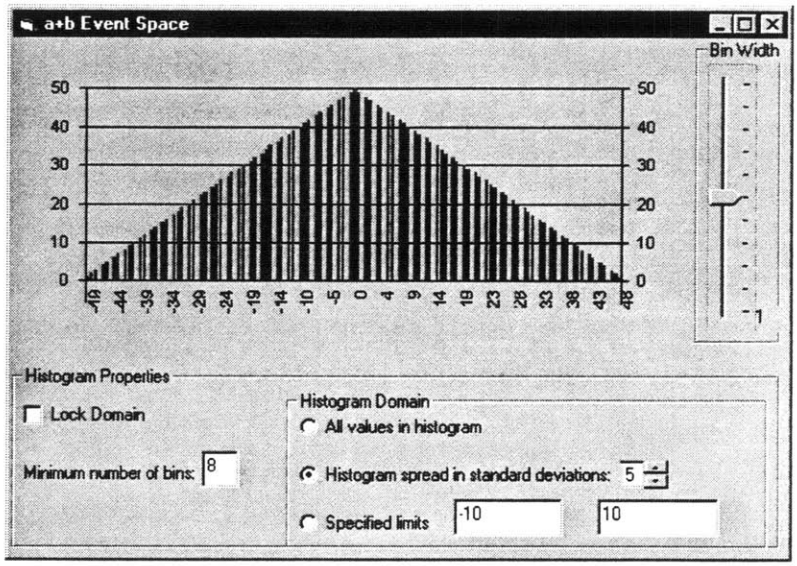

Figure 135: Event Space that Results from Adding Two Values with Uniform Event Spaces.

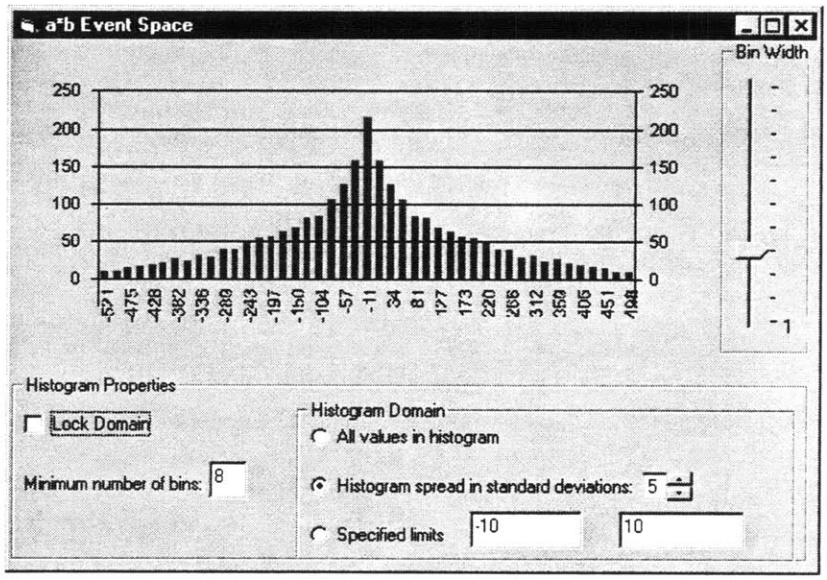

Figure 136: Event Space that Results from Multiplying Two Values with Uniform Event Spaces.

The differences illustrated in Figure 133 and Figure 134 reveal themselves in more familiar ways in the critical sensitivity of the results of mathematical calculations to statistical dependence between the arguments to those calculations. Figure 21 illustrates this phenomenon for a set of elementary expressions.

Unfortunately, the importance of inter-value dependency information demonstrated in Figure 21 is not at all exceptional: Typical programs of any type are filled with myriad dependencies. Some of the strongest dependencies reflect close relationships between the inputs to the program, or the variables within the program. (For example, we would expect the character of the individual components of a circuit input to a circuit simulator to be highly dependent. Certain exogenous inputs to a system dynamics model of the macroeconomy would likely exhibit correlation as well.)

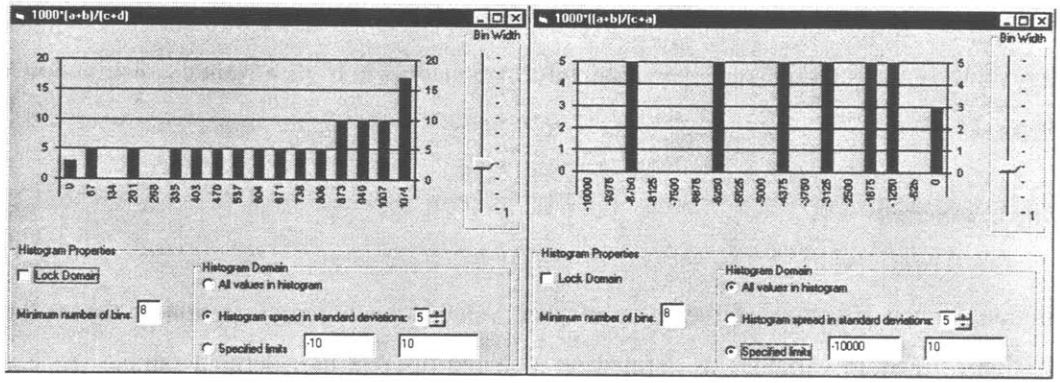

Figure 137: Comparison of the Event Space Results of Similar Expressions with Values Exhibiting Different Degrees of Internal Dependency

Appendix 2 noted the prevalence of dependencies in programs, and described three major classes of programs which exhibit large amounts of inter-value dependencies: Programs modeling physical systems, programs characterized by large amounts of control-flow dependence, and programs performing significant iterative calculations. In all of these cases, the user of event-space value modeling leads to the loss of very large amounts of information concerning patterns of program behavior. [Taylor 1995] [Osgood 1993] As noted in Appendix 1, given the particularly poor quality of the approximations to the results of value combination that are afforded by event-space representations, it seems dubious whether the benefits of modeling partially known values with such representations are sufficient to justify their costs.

11.4.1.4 Conclusion

This section has presented a brief introduction to the ideas and some of the motivations behind the use of the sample space value domain. In particular, the section first reviewed the idea (introduced in Appendix 2) of modeling program values as *random variables* defined over a sample space of possible execution contexts. The section then drew a distinction between representations of partially known values based on *sample space* characteristics of a program value distribution, and those characterizing value distributions according

to features of their *event space* (e.g. the maximum and minimum values). The discussion noted that event space representations invariably suffer from poor quality approximation when abstractly interpreting value combinations, even when the event space characteristics of the values being combined are precisely known. This loss of precision arises from the discarding of value *context* information that is implicit in the mapping of the random variable over the sample space, and seriously lessens the desirability of modeling partially known values using event space representations.

Having discussed some of the shortcomings of event space representations, we turn now to examine another strategy for value representation, one based on information drawn from a value's distribution over its sample space.

11.4.2 The Abstraction

11.4.2.1 The Basics

The inspiration for the structure of the sample space value domain representation is the mathematical characterization of random variables as defined on a shared discrete user-defined sample space. Like all random variables, every instance of the sample space representation defines a mapping from configurations in the sample space to values. This mapping defines a concrete[58], precisely known value associated with the program value for different possible probabilistic events (in this case, particular execution contexts). As for all sample spaces, each such configuration is associated with a particular probability of occurrence. All values in the program share the same set of possible execution contexts, and thus the same sample space.

It is important to stress that the configurations specified in the sample space represent (by definition) the finest-grain characterization of the possible probabilistic contexts (and thus of the possible program execution contexts). If there is a particular probabilistic event (in this case, an execution of the program with respect to certain particular parameters), it is guaranteed to be associated with one and only one instance of the sample space. Any two probabilistic events in sample space represent either precisely the same event, or completely disjoint events.

Unfortunately, the set of possible execution contexts associated with a particular program analysis can be very large – and sometimes unbounded. It is not feasible to require all program values to carry around complete mappings from each possible configuration in this space to specified program values. Fortunately, it is possible to lessen the computational requirements on value representations. The next two sections discuss techniques designed to reduce the cost of the representation and permit modeling in a wider variety

[58] Note that this value can additionally take on the distinguished elements $\top$ and $\bot$.

of situations. Although such techniques may appear to merely represent implementation strategies, they serve as refinements to the abstract that fundamentally change its capabilities and applicability.

In many cases it may be convenient to characterize the sample space configurations in a multidimensional manner. This is not an inherent aspect of the sample space itself, but merely a conceptually appealing way to describe the points in the sample space. Each element in the sample space – identifying a particular distinguishable outcome of the probabilistic experiment in question – can then be identified by a particular set of index values, with one such value specified for each of the sample space dimensions. Frequently the different dimensions of a sample space described in this way are associated a distinct source of modeled uncertainty.

11.4.2.2 Opportunities for Lossless Compression: Abbreviated Sample Space Representations

This section presents a mechanism that permits more compact representations of a variable's mapping from sample space configurations to values while maintaining the freedom for the user to enlarge those sample spaces at any point during analysis.

11.4.2.2.1 Background: Sample Space Characteristics of Program Values

As noted above and discussed in Appendix 1, program values (including inputs) within a program can be described as random variables defined on a unified sample space. These random variables differ in the pattern of values that they attach to each element of the sample space.

It was noted above that frequently sample spaces will be conceptualized as containing multiple dimensions, with each of the dimensions associated with some aspect of the execution context. Many values (random variables) within the program will depend only on a small set of the factors that form the dimensions of the sample space. The limited character of this dependence will reify in the mapping of sample space elements to values, and will in particular influence how the values assigned to particular sample space elements will vary.

For example, between elements of the sample space which are separated along dimensions associated with factors of the execution context to which the value (random variable) is sensitive, we would expect that the random variable would associate different values. Conversely, we would expect that a value (random variable) would not distinguish between (i.e. would retain uniform values between) two elements from the sample space that differ only with respect to aspects of the model that have no bearing on the value.

Taken in conjunction with the fact that most random variables associated with program values are likely to depend on a relatively small set of factors in the uncertainty model, the above observations imply that most

program values are likely to be associated with fairly regular sample spaces. More specifically, we would expect many random variables associated with program quantities to exhibit variation relative to a few dimensions of a large multidimensional sample space, but to be uniform over most other dimensions.[59]

11.4.2.2.2 Opportunities for Abbreviation

The regularity of the program value sample space suggests that representing the entire mapping within each variable is unnecessarily wasteful. As noted in the previous section, values assigned by a random variable to elements in sample space will frequently remain identical over many dimensions of that space. Suppose that this is true for dimension i of a space with n dimensions, and that function $f(x_1, x_2, x_3, x_4, \ldots, x_i, \ldots, x_n)$ represents the assignment of values to the sample space by the random variable. Here the space is effectively of dimensionality $n-1$, and we can define a function f' of dimension $n-1$ to represent the same space.

$$f'(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots x_n) = f(x_1, \ldots, 0, \ldots, x_n) = f(x_1, \ldots, x_i, \ldots, x_n) \; \forall \; x_i \text{ in the sample space}$$

More generally, suppose a program value P is associated with a sample space S of dimension n, with dimensions of size $a_1 \ldots a_{n-1}$, and a mapping function $M{:}S \rightarrow V$ that associates a value with each sample space configuration. Now suppose dimensions n-d..$n-1$ are uniform in value. That is, suppose $M(x_1, \ldots, x_{d-1}, x_d \ldots x_n) = M(x_1, x_2, x_3, \ldots x_{n-d-1}, 0, 0, \ldots 0) \; \forall \; x_k$ in the sample space

In this case, we can base the value instead on an abbreviated sample space S' of dimension $n-d$, and replace the mapping $M{:}S \rightarrow V$ with the equivalent but more compactly encoded $M'{:}S' \rightarrow V$, where $M'(x_1, \ldots, x_{n-d-1}) = M(x_1, \ldots, x_{n-d-1}, 0, 0, \ldots, 0)$.

Employing the same ideas in value representation allows us to encode a random variable solely in terms of those aspects of the value sample space to which the random variable is sensitive without the loss of any information in the process. Each element of M' that specifies a value for a specific point $(x_0, x_1, \ldots, x_{n-d-1})$ in S' in effect implicitly defines the values associated with $a_0 a_1 a_2 \ldots a_{d-1}$ sample points in S. A program value that is represented by M' can frequently be expressed much more compactly than an equivalent program value represented by M. In cases where a particular program value will depend on tightly circumscribed sources of uncertainty, this will represent a strong space savings.

---

[59] Note that while the arguments cited here may frequently hold, they are by no means guaranteed: It may be that a given program value depends on *all* factors (dimensions) of the execution context. To some degree, the uniformity observed here is artificial, depending as it does on the particular selection of dimensions by a user rather than entirely on some deeper characteristic of the value sample space. The selection of one basis set over another can have significant differences in terms of the compressibility of the space using the heuristic presented here: Through the application of a principle components transformation, what seems "uniform" in one case may be quite the opposite in another, and vice-versa.

While the use of an abbreviated sample space allows for more compact representation of a value without loss of information, the next section presents a technique designed to further drastically reduce the cost of maintaining value representations, but which achieves this aim only by discarding information.

11.4.2.3 Modes of Analysis

This section briefly describes the two styles of analysis under which the sample space representation can be used. The styles differ in their computational resource usage, in the contexts under which they are acceptable, and in the qualitative type of information that they offer the user. Most importantly for our focus here, they also differ significantly in the way that they are used most effectively.

The sample space approximating value domain described here permits the independent representation and simulation of values drawn from different execution contexts in a user-defined sample space. Analyses manipulating such values proceed as if they are being independently conducted on the execution context associated with each potential configuration. For programs associated with discrete sample spaces, the system is capable both of exhaustive and of probabilistic modeling of the possible paths of program execution. A brief introduction to each of these analysis regimes is given below.

11.4.2.3.1 Exhaustive Modeling

When exhaustively modeling sample space, program analysis is simultaneously conducted along *all* of the possible execution paths associated with possible program contexts. That is, the program is analyzed with information drawn from every possible execution context, no matter how large that set may be. As long as the collecting abstract domains are implemented in a safe manner, the information deduced by an exhaustive analysis is guaranteed to be conservative. That is, the information gathered during analysis is sure to represent a sound approximation to the information that would be deduced by *any* path of execution through the program at runtime. What this means intuitively is that we can be *certain* that the *event* space profile reported by program analysis is that which would result from independently executing the standard semantic program under each possible input context.[61] As discussed below, this certainty can offer important benefits in certain contexts and for certain forms of analysis.

By virtue of its guarantee of collecting analysis information on every possible execution context, exhaustive analysis essentially permits the user to make use of the sample space and the analysis process specifically as a means of performing *simultaneous analysis*. In this mode of analysis, the goal is typically to obtain a

[60] Continuous sample spaces – such as the set of all possible translations of an object in a continuous 2D space – are by nature uncountably large, and therefore not amenable to exhaustive modeli

[61] This assumes that the sample space characterization of the inputs is accurate.

cross-sectional view of analysis information collected across all points in the sample space. Because all possible input configurations are analyzed, it is possible to precisely examine and trace the evolution of each possible path of execution. Maintaining such a view during and following analysis helps foster an understanding of how the collected information varies with respect to the various (user-defined) dimensions of the sample space, and gives a definitive portrait of the *complete* range of analysis results that are possible within that sample space.

An example of where such a view might prove useful might occur in the context of a program that calculates various electrical characteristics (e.g. maximal current drain or voltage swings) associated with a circuit specified by the user. Suppose that in a particular circuit to be simulated that there slots for five types of components to be inserted. Suppose that any one of these slots could accept any of 4 pin-compatible components that are created by different manufacturers and that are potentially associated with different electrical characteristics. The set of possible inputs to this program is the set of possible circuit specifications – a domain of infinite size. But the sample space associated with the set of particular execution contexts of interest is rather compact, and distinctly finite. In particular, the sample space has 5 dimensions (one associated with each component that can be plugged in) and contains 4^5=1024 configurations, a small enough number to permit exhaustive analysis of the circuit simulation program to be conducted with respect to the entire space. This analysis could be used to show a cross-sectional view of a particular set of electric signals over time across any desired subset of the different possible configurations, to compare the range of electrical parameters and signal levels for the different possible circuits, and as a guide in tracking down causes for anomalous or undesirable electrical behavior by permitting the identification of the precise subset of possible inputs that demonstrate this behavior. By examining all possible configurations simultaneously, it is possible to weed out individual circuit configurations that might cause difficulties, and to identify those associated with the best quality characteristics.

11.4.2.3.2 Probabilistic Modeling

11.4.2.3.2.1 INTRODUCTION

While performing analyses that take into account all possible input configurations is a common ideal, it is frequently impractical to conduct such analyses. For example, many real-world situations involve sources of uncertainty that are continuous rather than discrete. Any discrete model sample space that attempts to model such uncertainty inherently represents an approximation; even if an analysis is conducted to examine all possible configurations drawn from such a discrete sample space, it will still fail to simulate an uncountably large number of possible physical configurations.

In other cases, the discrete sample space may be too large to have the program analysis realistically examine every potential configuration. In such cases it is also necessary to forego the analysis for certain possible configurations.

The alternative to exhaustive program analysis that is presented below is a form of *probabilistic* analysis. The adoption of probabilistic analysis has a variety of consequences both for what analysis tells us and for the manner in which analysis is conducted. In particular, because not all possible execution configurations are explored *in toto*, the results gathered by probabilistic analysis are not guaranteed to be correct for the entire sample space of possible execution contexts. Instead, they merely represent an approximation to the analysis results for the entire set of execution contexts. As will be discussed later, this has varying consequences for different sorts of collecting domains. What concerns us here is the fact that probabilistic modeling changes the manner in which analysis is conducted.

Some of the strongest uses of exhaustive analyses lie in delineating the *complete* range of possible analysis results for a given set of execution contexts. Probabilistic analyses cannot be employed for this purpose: While approximative ranges for analysis results over the sample space can be obtained probabilistically, there is no guaranteeing that a particular sample space configuration that is omitted from consideration will not lie outside the observed range. Similarly, one cannot use probabilistic results to compare complete cross-sections of analysis results defined across *all* execution contexts for a given point in the analysis. There are many types of collecting domains for which the only acceptable option is to collect complete and conservative information on the program operation. For such cases, probabilistic analyses cannot be used.

11.4.2.3.2.2 Opportunities for Probabilistic Analysis

However, probabilistic analysis does offer a number of strong benefits when used properly and in conjunction with collecting domains that tolerate approximation. In general, the types of applications to which it is most naturally suited are those that model the imprecisely known program values as probability distributions with collective properties, rather than as a set of individual configurations in which particular elements are examined. For example, in certain contexts, it may be desirable to examine the response of a piece of numerical code or a dynamical system to input consisting of perturbations around one or more important central points of interest. In this form of analysis, the goal is not to examine the output of the system for each possible output, nor to identify those unique configurations which are associated with the most important characteristics (e.g. yield the largest variation in system output, are associated with the largest signal values, etc.). The aim here is user *understanding* – to acquire a qualitative feel for the range of program behavior that is induced by input of a certain range. If a software designer can see how the system

maps a set of slightly uncertain inputs then he or she may be able to get a better understanding for how the system operates and for how different input varying in more general ways would be treated.

Examples of the valuable use of probabilistic analysis might include its use in investigating the operation of a system dynamics [Forrester 1968] model. In particular, we might be working with a model of a higher educational institution, and are interested in investigating the qualitative manner in which changes to policy or environmentally determined variables (e.g. the function defining the admissions and financial aid criteria, the amount of federal funding available, the amount spent on advertising the institution) end up affecting the visible characteristics of that institution (e.g. student headcount, the number of graduates and dropouts, or the total amount of financial aid awarded). We can begin by defining a suitable sample space for the analysis, in order to express the range of policies/environmental conditions that we wish to explore. Because the sample space in this case is for all practical purposes continuous, it is not feasible to have analysis completely explore the complete set of possible policy options. We can instead define a probabilistic distribution of samples around the current operating point, each differing slightly from the current values of the policy variables. By performing the analysis and examining the qualitative range and character of variation in values throughout the analysis, we can get a sense of the strength and character of the dependence on various system values to the input values.

11.4.2.4 Opportunities for Lossy Compression

11.4.2.4.1 Sample Space Downsampling: An Introduction

Although the representation of abbreviated sample spaces in place of full sample spaces represents a fundamental savings in representational cost, the size of the abbreviated sample spaces can still be enormous. One manner of reducing the costs associated with representation of values in the abbreviated sample space is to abandon the ideal of representing the entire space, and to opt instead for a sparse representation. The hope is that by judiciously choosing the sample points to discard and the times at which we downsample we will be able to maintain a good approximation to the behavior of the program over all of the proposed range of inputs while strongly reducing the cost of that analysis.

It is worth distinguishing between downsampling methodology and techniques that would attempt to lower value cost by transparently approximating the values held for samples in sample space (e.g. by means of interpolation or lossy compression of the value space). Two of the most important differences are listed below:

- **No Distortion of Individual Samples.** In contrast to strategies that would permit imprecise information to be associated with particular samples, the quantum of knowledge using the sample space

representation is the precise value for a particular sample. That is, the downsampled sample space representation always maintains either full information or no information whatsoever concerning the value of a sample. If the analysis returns any information a particular sample, that information is guaranteed to be correct. In contrast to strategies which would approximate the values associated with particular points in sample space, the only inexactness associated with the downsampling methodology lies in the domain of the event space. In particular, while the information regarding particular samples in sample space is always precise, the measurements of the properties of the sample space may be inaccurate. For example, by dropping certain samples from the sample space, we may distort the range of apparent variation of the random variable.

- **Awareness of Imprecision.** The loss of information in downsampling is not transparent: Sample elimination has very pronounced and long-term effects for the surrounding sample space. Whenever manipulating a value, it is a simple matter to get a very precise measure of what fraction of the values from sample space have been discarded and to assess the probabilistic consequences of this loss. Algorithms making use of downsampling may not always have access to the maximally precise collective information concerning a value, but they can always be aware of the extent of their ignorance.

The particular strategy adopted for downsampling has a number of important consequences and tradeoffs, some of which are surveyed in the next section. The section beyond that continues on to discuss some of the principles that can be applied to guide the selection of values to discard.

11.4.2.4.2 Consequences of Downsampling

The use of downsampling to reduce the costs of value representation has far-reaching consequences for the semantics and practical use of program values. Some of the most important (and obvious) of these effects are listed below:

- ***Non-Conservative Analysis.*** For certain categories of model uncertainty, the sample space representation can exhaustively describe all possible states for a value. Under such situations, the behavior of a program analysis is guaranteed to represent a conservative approximation to the run-time behavior of the program being analyzed on the *entire range of possible uses of the program.* The fact that an analysis is conservative in this sense frequently permits very strong action to be taken. (For example, optimizations may be made to program structure that would not in general be legal but which are acceptable under the particular observed conditions of program execution, since it is certain that those conditions safely approximate the *entire range* of program behavior. Similarly, the user can be sure that a collecting domain reporting reaching definitions for a given point has indeed reported *all* such definitions that are possible.)

If in the interests of minimizing resource usage the system allows the elimination of certain sample values from the sample space, it is in essence avoiding the consideration of program behavior for certain categories of program input. Under such conditions, we can no longer be sure that an analysis has simulated approximations to all possible ranges of program operation, and that the collecting domains have indeed accumulated information on all possible avenues of program operation.

It is important to note, however, that the loss of certainty that accompanies the use of non-conservative representations is by no means *always* problematic. Both conservative and non-conservative value representations are united by the fact that they serve as *approximations* to the true value distribution. Conservative representations have the advantage of allowing complete confidence that they describe all possible results, but in the process of gaining this guarantee typically are forced to sacrifice substantial amounts of discriminatory power regarding the distributions associated with values. By contrast, the sample space representation cannot guarantee that it has accurately described all possible results of a computation, but may give an excellent indication as to what values are likely to occur. In short, conservative representations frequently sacrifice a great deal of probabilistic knowledge for the sake of a small amount of *certain* knowledge; stochastic representations can take the opposite approach.

As will be discussed in Section 11.4.2.3, the practical impact of this uncertainty varies greatly between varieties of collecting domains and regimes of user interest. In many cases the consequences are far less severe than one might have cause to expect, and the use of non-conservative representation can bring substantial benefits.

- ***Weakened Knowledge of the Value Distribution.*** The elimination of some of the samples means that there is a range of possible input values whose paths through the program are no longer being simulated. This can sacrifice precision in the output relative to what is possible given an exhaustive exploration of sample space. If we eliminate a great many of the samples from the sample space, it would seem reasonable to expect that there would be a correspondingly large loss in the precision and value of the analysis. Fortunately, such fears are frequently unfounded: Due to regularities and patterns that arise from the combinatorics of value combination and dependencies, resource-conserving elimination of samples in larger dimensional space frequently has far less effect on the execution dynamics than one might naïvely expect. In most situations, we can discard samples without qualitatively affecting the accuracy of the value distribution.

- ***Samples Space Equivalence No Longer Guaranteed.*** The motivation behind maintaining a value representation based on models of the program sample space is to enable the independent simulation of

program values drawn from different potential program inputs. The hope here was to allow values drawn from a particular execution context to be combined only with other values drawn from the same context. If the analysis process is permitted to conserve computational resources by discarding samples, two values being combined will not *necessarily* contain the same residual sample space values: Some of the values discarded from one value may still remain in the other value. Because the values maintained for a value by neighboring positions in sample space in general need not relate to one another in any close or meaningful way, it does not seem realistic to contemplate "interpolating" a value for a missing sample point. The consequences of the alternative (discarding any sample that appears only in one of the values to be combined) may not be entirely clear at this point.

While a full examination of the impact of sample space downsampling on the amount and quality of information carried during analysis must be delayed until Section 11.4.3, it should be clear that the technique offers definitive savings in terms of the computational resources demanded by analysis. In particular, discarding samples allows for smaller per-value memory consumption as well as more rapid value combination.

11.4.2.4.3 Downsampling Methodology

This section describes the factors taken into account when deciding which values to eliminate from a sample space representation. This description does not yet consider the implementation issues related to downsampling (particularly the way in which downsampling is handled by value combination machinery). Discussion of these topics will be delayed until later in this chapter.

The fundamental tradeoff addressed by downsampling is that between the amount of information (and thus precision) maintained on runtime values, and the computational resources consumed (both space and time). An ideal downsampling methodology would be one that tied downsampling to a cost-benefit analysis for the value at hand. The primary difficulty facing the use of such a methodology for downsampling is the fact that it is difficult to accurately anticipate the benefits derived from a particular value representation.

The first-order *cost* of maintaining additional samples within a value representation is frequently clear from a cursory preliminary analysis of the program text. These costs lie primarily in the expense of the space consumed by the value representation itself, as well as the larger time needed to combine together values containing more sample points. For a particular size of acceptable value representation, there will typically be certain patterns of downsampling that admit to more efficient implementations and more compact representation.

By contrast, the *benefits* associated with maintaining an additional sample are less obvious. Two important considerations are listed below, in order of their strengths as guides to the downsampling process.

- **Likelihood of Runtime Occurrence.** Each element in sample space is associated with a probability that this configuration will occur at runtime. Analysis information collected about more likely configurations provides more information concerning runtime behavior than does analysis information regarding less likely configurations.

- **Likelihood of Analysis Use.** Although downsampling affects the character and mechanics of value combination in a significant way, an overview of the value combination process must wait until the discussion of value dynamics in 11.4.3. Nonetheless, there is one aspect of that process which is sufficiently important to downsampling tradeoffs to be worth discussing here. In particular, in weighing the benefits of representing a value for a particular configuration in sample space it should be understood that this information will not be used in every combination with other values. The reason for this is that in order for an operator to compute a value for a particular element in (complete, not abbreviated) sample space, the value associated with that element must be known for *every* program values being combined by that operator. Phrased in another way, in general the value of a particular sample space configuration must be known for *all* of the values being combined in order to be known for the result of that combination. What this means is that knowledge of the value of a particular sample space element within a particular program value will not offer any accuracy advantages to analysis unless that program value is combined with other program values that are also carrying information for that element. A consideration of this fact implies that it would be desirable to coordinate any downsampling performed on different program values in such a way that the downsampling tends to eliminate similar sample values or preserve similar samples between the values.

The list above considered a pair of factors that can be used in heuristics to aid in decisions as to which samples to eliminate from a program value representation when downsampling. The use of information concerning the relative frequencies of sample space configurations is particularly desirable in that the information is readily available and permits the formulation of very simple heuristics. The other two considerations (taking into account the likely extent of analysis use of particular samples, and making use of

[62] It should be stressed that this rule applies only at the level of the *unabbreviated* sample space. As discussed in the previous section, a particular sample defined in an "abbreviated sample space" in general represent a subspace of samples associated with equivalent values within the complete sample space. It is possible that the coordinates of two such samples drawn from different program values could differ while the subspaces overlap. There are more complex but equivalent criteria that can be used to determine if a given sample in the abbreviated sample space will be used when the containing program value is combined with another program value.

information on the position of samples within the distribution of possible values) require greater thought and implementation effort.

11.4.2.5 Summary

This section has examined two techniques that can be used to reduce the size of the sample space representation for program values. Each of these techniques changes the character of the representation in significant ways: The maintenance of "abbreviated" sample space representations permits the use of sample spaces of much higher dimensionality, providing that values based on those spaces vary only with respect to few dimensions of the space. By contrast, the use of sample space downsampling permits the replacement of exhaustive modeling of all program behavior by imprecise but far less costly modeling of a wide but non-exhaustive range of program execution. The downsampling approach, however, changes the semantics of analysis in very important ways, such that the event-space results from analysis are no longer a conservative approximation to program behavior under *all* possible execution contexts, but only under a selected subset of the possible contexts.

Having examined some of the issues in modeling program values in sample space, we turn now to a discussion of the manners in which analysis results are used and the conditions under which a downsampling methodology may be desirable.

11.4.3 Combinatorics of Value Combination

This section briefly examines the combinatorics that obtains for value operations. We begin with simple mathematical model of the value combination process that demonstrates serious problems that can arise with a naïve downsampling strategy. We then discuss a solution to this difficulty that requires no changes to the value combination machinery.

11.4.3.1 Naïve Combinatorics

11.4.3.1.1 Fundamental Algorithm

This section investigates the statistics associated with combining two program values represented using the sample space domain. In particular, we examine how the composition of the sample sets in the values being combined affect the value that results from the combination.

The characterization of value combination in the full context of sample space is simple, and is shown in Figure 138. The value associated with each coordinate of the sample space for the value result is simply the combination of the values at corresponding coordinates in the values being combined.

```
For each coord in sample space
    v_result[coord] = v1[coord] ⊕ v2[coord]
```
Figure 138: Value Combination in Full Sample Space

The use of the "abbreviated sample space" representation discussed in Section 11.4.2.2, however, complicates matters. In particular, each value being combined may be vary over (and thus be expressed relative to) different dimensions in the global sample space. From each of these two values, we need to be able to create and populate a new abbreviated sample space, each of whose members combines the appropriate values in the sample spaces of the arguments. The characteristics of the abbreviated sample spaces that are relevant to our investigation are the identity and size of the dimensions. By the rules established in Section 11.4.2.2, if a particular global sample space dimension is not specified in a value's abbreviated sample space, the value must be uniform across that dimension. Given this consideration, suppose that we are combining two values v_1 and v_2 that are represented using the samples space representation. Suppose further that v_i is associated with n_i samples and D_i dimensions, of which s are shared. Without loss of generality, suppose the shared dimensions are called $d_{i,1}\ldots d_{i,s}$, and the non-shared dimensions are $d_{i,(s+1)}\ldots d_{i,D1}$. Figure 139 shows the pseudocode required to perform the value combination.

```
Create sample space value v_result with dimensions { d1,1…d1,D1 } ⋃ { d2,1…d2,D2 }
coordsShared1 = { d1,1.min(), d1,2.min(),… d1,s.min() }
while coordsShared1.Increment()
    coordsUnshared1 = { d1,(s+1).min(), d1,(s+2).min(),… d1,D1.min() }
    coordsUnshared2 = { d2,(s+1).min(), d2,(s+2).min(),… d2,D1.min() }
    while coordsUnshared1.Increment()
        while coordsUnshared2.Increment()
            v_result [coordsShared1, coordsUnshared1, coordsUnshared2] =
                v1[coordsShared1, coordsUnshared1]
                ⊕ v2[coordsShared2, coordsUnshared2]
```
Figure 139: Value Combination in Abbreviated Sample Spaces.

The abbreviated sample space resulting from the value combination varies over (and is thus expressed over) the *union* of the set of dimensions included in the abbreviated sample space of the values being combined. The values in the resulting sample space will typically vary with respect to more dimensions than the values in the operand sample spaces, and the algorithm above will have to populate the value at each such coordinate position in the new abbreviated sample space.

11.4.3.2 The Infeasibility of Random Downsampling

The algorithm shown in Figure 139 operates in the absence of downsampling. Handling downsampling in the operands and the resulting value requires substantially more mechanism, including reasoning about the cost/benefits associated with downsampling different dimensions.

Before turning to a discussion of the sample space implementation of the approximating value domain interfaces (including survey of how downsampling is currently implemented), we first consider why downsampling cannot be performed *randomly*. In particular, we characterize the results of combination in the context of random downsampling and demonstrate its infeasibility.

Recall from Section 11.4.2.3 the sample space resulting from a value combination will contain a value for a particular (global) coordinate only if values are specified for that coordinate in *all* of the values being combined. This consideration is sufficient to allow us to calculate the number of expected samples in the resulting value using a *randomly* downsampling version the algorithm shown in Figure 139. In particular, Figure 140 calculates the number of samples expected in the output space resulting from combining two values *before* downsampling by summing up the expected number of pairs over samples combined for each possible coordinate position in the shared sample space. We then demonstrate that even absent any downsampling of the resulting value, this strategy will yield far too few values in the resulting sample space to be feasible.

If we make the key assumption that the samples in each of the values being combined are independently and uniformly distributed within their (abbreviated) sample spaces, then we can model the distribution of samples within a given bin for the output value as the outcome of a Bernoulli process. The probability p of success for this process is equal to the reciprocal of the size of the abbreviated output sample space. Proceeding more formally, we arrive at the calculation shown in Figure 140. This calculation indicates the number of samples that are expected from a combination of two values with samples randomly scattered throughout sample space. Figure 140 demonstrates that the resulting value will on average contain a number of samples equal to the product of the number of samples in each of the operands divided by the volume of the *shared* dimensions.

Suppose that we have two values, v_1 and v_2.

Suppose further that v_i contains shared dimensions $\{ d_1,...d_s \}$ and unshared dimensions $\{ d_{i,s+1},...d_{i,D_i} \}$

Then the resulting space S will include dimensions $D_S = \{ d_1,...d_s, d_{1,s+1},...d_{1,D_1}, d_{2,s+1},...d_{2,D_2} \}$

Given that v_1 and v_2 have been randomly but independently downsampled, we can compute the probability p that a particular point $\{ x_0, x_1, x_2,...x_{D_S} \}$ will be present in the resulting space S.

In particular,

p = Probability that v_1 contains a point $\{ x_0, x_1,...x_S, x_{S+1},...x_{D_1} \}$ and

v_2 contains a point $\{ x_0, x_1,...x_S, x_{S+1},...x_{D_2} \}$

$$p = \prod_{j=1}^{2} \left(\frac{|v_j|}{\prod_{i=1}^{s} |d_i| \prod_{i=s+1}^{D_1} |d_{j,i}|} \right) = \frac{|v_1||v_2|}{\left(\prod_{i=1}^{S} |d_i| \right)^2 \prod_{i=s+1}^{D_1} |d_{1,i}| \prod_{i=s+1}^{D_2} |d_{2,i}|}$$

Thus the expected number of points in the resulting space S is just p times the number of elements of S:

$$E(|S|) = p \cdot \text{Volume}(D_S) = \frac{\prod_{i=1}^{s} |d_i| \prod_{i=s+1}^{D_1} |d_{1,i}| \prod_{i=s+1}^{D_2} |d_{2,i}| |v_1||v_2|}{\left(\prod_{i=1}^{S} |d_i| \right)^2 \prod_{i=s+1}^{D_1} |d_{1,i}| \prod_{i=s+1}^{D_2} |d_{2,i}|} = \frac{|v_1||v_2|}{\prod_{i=1}^{s} |d_i|}$$

Figure 140: Calculations to Determine the Number of Values Expected After a Combination of Two Independently, Randomly Downsampled Values.

If we further assume that the two operands contain the same number of samples and that we would like the resulting value to have the same sample count as well, the formula derived in Figure 140 forces an uncomfortable conclusion that is spelled out in Figure 141. In order to maintain the same number of samples in the resulting value as are in the input, we would need the number of samples present in the input operands to approximate the size of the sample space – a potentially enormous number.

Now, suppose $|v_1| = |v_2|$

if we want the number of samples in the resulting value to equal the number of samples in the operands, then we want

$$\frac{|v_1||v_2|}{\prod_{i=1}^{s}|d_i|} = \frac{|v|^2}{\prod_{i=1}^{s}|d_i|} = |v| \Rightarrow |v| = \prod_{i=1}^{s}|d_i|$$

Figure 141: Implications of the Combinatorics of Random Downsampling.

Even when the requirements for sustainable combination given in Figure 141 are met on average, there is still no assurance that they will always be met. Moreover, within analysis even a single value associated with low sample count can force all subsequently combined samples to a lower sample count. While the random sampling technique has the virtue of simplicity, it places the system at risk of returning sample space values associated with decreasing size over time.

The shortcomings of the random selection downsampling approach dictated the selection of a more structured downsampling strategy that ensured a more sustainable supply of samples. For further information on the approach taken, the interested reader is referred to Section 11.4.4.4 and the complete code at the location given in Appendix 3.

11.4.4 Implementation

11.4.4.1 Introduction

The sections above have described the sample space approximating value domain abstraction and the abstract model to which it adheres. In adherence with to the framework used for previous approximating domain presentations, this section examines the implementation of the sample space domain. It should be kept in mind that even more so than for the other domains, the code that is shown is only a small fraction of the total code required for implementation.

11.4.4.2 Preliminaries

11.4.4.2.1 Subdomain Classes

The sample space domain is implemented as a common superclass (*QuadDomainGuidingValue*) subclassed by four different subclasses specialized for particular categories of domain elements. In particular, the subclass *ConcreteQuadDomain* is used to represent concrete element (elements that are uniform across all execution contexts), *BottomQuadDomain* for the bottom element, *TopQuadDomain* for top elements, and

BUQuadDomainGuidingValue for values disaggregated by execution context.[63] In the section below, attention will be focused on the operation of the sample space values. We adopt this focus on account of the fact that it is by the presence of such values that the current domain is distinguished, and because the vast majority of the code used to implement the domain is used to represent the disaggregated values. For the most part, the presentation will follow the conventions set in earlier sections.

11.4.4.2.2 Implementation of the Sample Space Model

The central component of the *BUQuadDomainGuidingValue* implementation is the specification of the values (drawn from a 3-level domain) associated with each coordinate in the "abbreviated sample space" (Section 11.4.2.2). In addition, the *BUQuadDomainGuidingValue* class is associated with a maximum sample size. If the result of a calculation is found to require more than this number of samples, the sample space will be lossily "downsampled" (Section 11.4.2.3) to allow for approximate representation of the value. (Note that downsampling can be avoided altogether by setting this value larger than the size of the full sample space).

Downsampling is carried out in an incremental manner (by powers of two) and on a per-dimension basis. This approach guarantees substantial overlap in the retained samples between values in which a dimension has been downsampled and those in which no or lesser downsampling has taken place. The downsampling is triggered by recognition that the sample count resulting from a value combination will exceed the allowable sample count. In those cases where downsampling is required, the system selects a particular dimension for downsampling based on certain cost/benefit criteria, and eliminates all samples occupying every other coordinate along that dimension. Thus, even a value in which a dimension has been downsampled n times will exhibit substantial overlap with any other value containing that dimension. In certain contexts, it is judged desirable to completely collapse a dimension to just a single coordinate; the dimension is henceforth known as a "sparse" dimension of the sample space.

The structure of the abbreviated sample space (*e.g.* the dimensions that comprise it) and the mapping from that space to values is encapsulated in the *IntSampleSpace* class. *IntSampleSpace* serves as the central abstraction for representing execution-context-disaggregated values. The *IntSampleSpace* representation also keeps track of the downsampling applied to each axis of the sample space, and performs the cost/benefit analysis necessary to identify which sample space dimension is most suitable for collapse or (more drastically, for representation as a "sparse" dimension).

[63] The "BU" stands for "Bounded Uncertainty."

11.4.4.3 Element Creation

The creation of elements in response to value initialization or the appearance of manifest constants in the code is handled by the *QuadDomainGuidingValue* superclass, as shown in Figure 142. A request to create an abstract domain element from a constant simply returns a new *ConcreteQuadDomain* value associated with the appropriate uniform value. The creation of an uninitialized value yields the distinguished domain top element, as represented by an element of the *TopQuadDomain* class.

```
public _IIntDomainInfo CreateFromConstant(int v, int srcLoc)
{
    return(s_MRGuidingValueIntReturnValue = new ConcreteQuadDomain(v));
}
public _IIntDomainInfo CreateUninitialized(int srcLoc)
{
    return s_MRGuidingValueIntReturnValue = s_topDomainElt;
}
```

Figure 142: Handling of Domain Element Creation

11.4.4.4 Element Combination

The combination of sample space values requires coordination between many distinct levels of abstraction. The *BUQuadDomainGuidingValue* abstraction implements the *operatorAdd* method required by the *_IIntDomainInfo* interface as shown in Figure 143. The *operatorAdd* method calls off to a more general method (*ProcessBinaryOperator*) implemented by the *QuadDomainGuidingValue* superclass, passing in an object that will implement the addition-specific value combination behavior (e.g. recognizing instances where zero elements are being added to values and combining concrete values via addition).

```
static {
    s_addOperator = new addIntBinOp();
    ...
public class addIntBinOp extends IntBinOp
{
    public addIntBinOp()
        {
        fHasLeftAnnihilator = fHasRightAnnihilator = false;
        fHasLeftIdentity = fHasRightIdentity = true;
        leftIdentity = rightIdentity = 0;
        }
    public int specifiedCombineStandardSemanticsValues(int  v1, int  v2) { return(v1 + v2); }
}
public _IIntDomainInfo operatorAdd(_IIntDomainInfo arg2)
{
    IntBinOp tmp = s_addOperator;
    return(this.ProcessBinaryOperator(tmp, arg2));
}
```

Figure 143 : The Handler for the Integer Addition Operator in the Sample Space Domain.

```
    private _IIntDomainInfo ProcessBinaryOperator(IntBinOp op, _IIntDomainInfo arg2)
        throws IllegalArgumentException
    {
```

```
        QuadDomainGuidingValue v1 = this;
        QuadDomainGuidingValue v2 = (QuadDomainGuidingValue) arg2;

        switch (v1.QuadDomainTag())
        {
        case ConcreteEpistemicFlavor:
            {
                ConcreteQuadDomain narrowedV1 = (ConcreteQuadDomain) v1;
                switch (v2.QuadDomainTag())
                {
                case ConcreteEpistemicFlavor:

    return s_MRGuidingValueIntReturnValue = op.Apply(narrowedV1, (ConcreteQuadDomain) v2);
                case BoundedUncertaintyEpistemicFlavor:

    return s_MRGuidingValueIntReturnValue = op.Apply(narrowedV1, (BUQuadDomainGuidingValue) v2);
                case UnknownEpistemicFlavor:
                return s_MRGuidingValueIntReturnValue = op.Apply(narrowedV1, (TopQuadDomain) v2);
                default:

    throw new IllegalArgumentException("Did not recognize second argument type in ProcessBinaryO
perator");
                }
            }
        case BoundedUncertaintyEpistemicFlavor:
            {
                BUQuadDomainGuidingValue narrowedV1 = (BUQuadDomainGuidingValue) v1;
                switch (v2.QuadDomainTag())
                {
                case ConcreteEpistemicFlavor:

    return s_MRGuidingValueIntReturnValue =  op.Apply(narrowedV1, (ConcreteQuadDomain) v2);
                case BoundedUncertaintyEpistemicFlavor:

    return s_MRGuidingValueIntReturnValue =  op.Apply(narrowedV1, (BUQuadDomainGuidingValue) v2)
;
                case UnknownEpistemicFlavor:

    return s_MRGuidingValueIntReturnValue =  op.Apply(narrowedV1, (TopQuadDomain) v2);
                default:

    throw new IllegalArgumentException("Did not recognize second argument type in ProcessBinaryO
perator");
                }
            }
        case UnknownEpistemicFlavor:
            {
                TopQuadDomain narrowedV1 = (TopQuadDomain) v1;
                switch (v2.QuadDomainTag())
                {
                case ConcreteEpistemicFlavor:

    return s_MRGuidingValueIntReturnValue =  op.Apply(narrowedV1, (ConcreteQuadDomain) v2);
                case BoundedUncertaintyEpistemicFlavor:

    return s_MRGuidingValueIntReturnValue =  op.Apply(narrowedV1, (BUQuadDomainGuidingValue) v2)
;
                case UnknownEpistemicFlavor:

    return s_MRGuidingValueIntReturnValue =  op.Apply(narrowedV1, (TopQuadDomain) v2);
                default:

    throw new IllegalArgumentException("Did not recognize second argument type in ProcessBinaryO
perator");
                }
            }
        default:

    throw new IllegalArgumentException("Did not recognize second argument type in ProcessBinaryO
perator");
```

```
        }
    }
```

Figure 144: The Generic Binary Operator Processing Method of the *QuadDomainGuidingValue* Class.

Figure 144 shows the central routine used to handle value binary combination for any operator and any epistemic types for the operands. Based on the epistemic classification for the operands, this method dispatches to appropriate routines in the operator abstraction. As shown in Figure 145, the operator abstraction simply forwards a call to combine sample space values on to the appropriate method in the sample space abstraction (in this case, *ApplyBinOpToBUBU* – a routine in *BUQuadDomainGuidingValue* that combines two sample space values).

```
    public QuadDomainGuidingValue Apply(BUQuadDomainGuidingValue argLeft, BUQuadDomainGuidingVal
ue argRight)
    {
        return argLeft.ApplyBinOpToBUBU(argRight, this);
    }
```

Figure 145: The Epistemic-Type-Specific Routine for the Binary Combination Operator.

The *ApplyBinOpToBUBU* routine in turn just computes the new cap on the number of samples allowed for the routine output, and then calls off to the top-level method for computing the output sample space – a routine called *ComputeCombinedSampleSpace* in the *IntSampleSpace* class.

```
public
    QuadDomainGuidingValue ApplyBinOpToBUBU(BUQuadDomainGuidingValue argRHS, IntBinOp  combiner)
{
    long
    ctResultMaxSamples = CtResultSamplesFromCombination(this.m_ctMaxSamples, argRHS.m_ctMaxSampl
es);

    IntSampleSpace resultSpace = this.m_sampleSpace.ComputeCombinedSampleSpace(argRHS.m_sampleSp
ace, combiner, ctResultMaxSamples);
    BUQuadDomainGuidingValue  buResult = new BUQuadDomainGuidingValue(resultSpace, ctResultMaxSa
mples);
    return buResult;
}
```

Figure 146: Method for Combining Two Sample Space Domain Values.

The algorithm to combine values from two abbreviated sample space spaces is relatively straightforward, but is complicated by the fact that each of the sample spaces may include different sets of domains, and even the domains held in common between the sample spaces may be downsampled by different amounts. Figure 139 in Section 11.4.3 characterized the sample space resulting from a combination of sample spaces using pseudocode; the routine *ComputeCombinedSampleSpace* in the *IntSampleSpace* (shown in Figure 147) implements that mapping.

The routine first determines the (abbreviated) sample space dimensions that are shared between each of the operands, as well as the sets of dimensions unique to each one of the operands. These sets of dimensions are passed along with the limiting size of the result set, the binary operator to apply to elements, to a method (*ComputeCombinedSampleSpaceFromDimensions*) responsible for actually creating and populating the new sample space.

```
public IntSampleSpace ComputeCombinedSampleSpace(IntSampleSpace rightSampleSpace, IntBinOp binOp
, long ctMaxSamples)
{
    SampleSpaceDimensions sharedDimensions = new SampleSpaceDimensions();
    SampleSpaceDimensions onlyArg1Dimensions = new SampleSpaceDimensions();
    SampleSpaceDimensions onlyArg2Dimensions = new SampleSpaceDimensions();

    this.m_dimensions.ComputeSharedUnshared(rightSampleSpace.m_dimensions, sharedDimensions, onl
yArg1Dimensions, onlyArg2Dimensions);
    return(this.ComputeCombinedSampleSpaceFromDimensions(rightSampleSpace, binOp, ctMaxSamples,
sharedDimensions, onlyArg1Dimensions, onlyArg2Dimensions));
}
```

Figure 147: Top-Level Sample Space Value Combination Routine of *IntSampleSpace*.

As shown in Figure 148, the routine *ComputeCombinedSampleSpaceFromDimensions* performs two major tasks. First, the output space is created with a set of dimensions consisting of the union of the set of dimensions in each of the operands. Secondly, the routine goes through and combines each pair of compatible samples within the sample spaces. Before examining the sample combination process (which is quite simple), we will examine the more complicated routine that takes care of the first of these responsibilities – creating the output sample space.

```
private IntSampleSpace ComputeCombinedSampleSpaceFromDimensions(IntSampleSpace rightSampleSpace,
 IntBinOp binOp, long ctMaxSamples, SampleSpaceDimensions sharedDimensions, SampleSpaceDimension
s onlyArg1Dimensions, SampleSpaceDimensions onlyArg2Dimensions)
{
//  allocate one of our type
    IntSampleSpace resultingSampleSpace = new IntSampleSpace();
    //  ok, create the resulting sample space, with the appropriate HybridDimensions.
    //  note that this calculates the sparse/exhaustive HybridDimensions

    resultingSampleSpace.CreateCombinedSpace(ctMaxSamples, sharedDimensions, onlyArg1Dimensions,
 onlyArg2Dimensions);
    //  establish context within the combiner

    binOp.EstablishCombinationContext(this, rightSampleSpace, resultingSampleSpace, sharedDimens
ions, onlyArg1Dimensions, onlyArg2Dimensions);
    // note that this is called on the resulting sample space

    resultingSampleSpace.CoreComputeCombinedSampleSpaceFromDimensions(binOp);
    return(resultingSampleSpace);
}
```

Figure 148: Method to Compute the Resulting Sample Space of a Combination from Knowledge of the Shared and Distinct Dimensions.

Most[64] of the method used to determine the structure of and create the sample space resulting from a combination (*CreateCombinedSpace*) is shown in Figure 149. This routine is responsible for structuring the output space in a manner compatible with the structure of both of the input operands and with the overall space requirement (embodied in the parameter *dMaxCost*). In particular, the routine must first observe the downsampling constraints by not assigning a smaller downsampling value for an output dimension that is smaller than a downsampling value for the same dimension in any operand. Secondly, the total number of samples of the output space (as computed by the volume of the space enclosed by all present dimensions divided by the aggregate downsampling ratio) must be less than the parameter *dMaxCost*.

While the user can effectively prevent downsampling through appropriately large choice of *dMaxCost*, the situation becomes more complicated when the output sample space is discovered to require downsampling. In such cases, the system performs a cost-benefit calculation to judge the relative net value of each dimension; the dimension that is judged least valuable is downsampled appropriately. In order to perform this calculation, the system uses information concerning each dimension. In particular, the system collections for each dimension information on:

- Whether the dimension is sparse (i.e. has only one sample) or disaggregated

- The minimum acceptable downsampling for that dimension. (Because a sample must be present in *all* values being combined to appear in the output, we must at the least omit from the output any values omitted from the values being combined. As a result, the minimum acceptable downsampling is the least common multiple of the downsampling rates for the dimension in the operands).

- The result of applying a simple cost function to the dimension, based on whether the dimension is shared or not. At present, this cost function simply returns the width of the dimension in samples when downsampling is taken into account, minus a tiny bias indicating whether the dimension is shared or unshared. The net result is that the system will preferentially downsample dimensions currently requiring many samples to represent, but will prefer the downsampling of *unshared* dimensions to shared ones.)

[64] Two initialization loops very similar to the one shown have been omitted in the interest of space and clarity.

Based on this information, the system progressively downsamples the most costly dimensions until the specified cost (sample)[65] threshold is reached.

```
// having the args in this form avoids worrying about duplicates
public void CreateCombinedSpace(double dMaxCost, SampleSpaceDimensions sharedDims, SampleSpaceDi
mensions onlyArg1Dims, SampleSpaceDimensions onlyArg2Dims)
{
    /* ok, we know what dimensions MUST be sparse here
    ' we have to decide which, if any of the dimensions that COULD be exhaustive should
    '   be collapsed to sparse dimensions
    ' the way we do this is to examine the total size of the space that would be created
    '   if we were to keep the maximal # of dimensions exhaustive
    ' if this space is too large, we successively collapse dimensions (largest first) until we g
et a smaller space
    '   if two dimensions are of the same size, we collapse the one that is not shared between t
he two arguments (perhaps it is less common) */
    DimIdCostCounts rgDims[] = new DimIdCostCounts[sharedDims.Cardinality() + onlyArg1Dims.Cardi
nality() + onlyArg2Dims.Cardinality()];
    int iRecorded = 0;
    int ctResultDownsamplingMin = 1;
    int i;

    for (i = 0; i <sharedDims.Cardinality(); i++)
    {
        rgDims[i].dimId = sharedDims.DimIdForDimIndex(i);

    rgDims[i].dCostBenefitRating = this.DComputeCostBenefit(sharedDims, i, 2);  //  if not in sh
ared then it is in 1 argument
        rgDims[i].fSparse = sharedDims.FSparseCoordPosition(i);

        iRecorded++;

    ctResultDownsamplingMin = GlobalUtility.LCMPowerOfTwo(sharedDims.CtDownsampling(), ctResultD
ownsamplingMin);
    }
    … Similar population of rgDims array by the non-shared dimensions…
    // ok, shrink the least desireable exhaustive dimensions until we can shrink no more or are
cheap enough
    int     iLargestCost;
    // we separate this out in order to allow customization
    int     ctResultDownsampling = ctResultDownsamplingMin;
    //  rgPseudoILargestCostOut[0] holds iLargestCost
    while ((this.DMeasureAggregateCost(rgDims, rgPseudoILargestCostOut, ctResultDownsampling) >
dMaxCost))
    {
        iLargestCost = rgPseudoILargestCostOut[0];
        // ignore the sparse dimensions, and shrink the largest exhaustive dimension
        if (iLargestCost == -1)
            ctResultDownsampling *= 2;
        else
            rgDims[iLargestCost].fSparse = true;
    }
    iLargestCost = rgPseudoILargestCostOut[0];
    … Fill in the arrays rgDimIds and rgFSparse with the results of the downsampling…
    this.m_dimensions.Create(rgDimIds, rgFSparse, ctResultDownsampling);
    …
    this.m_samples.CreateSamples((int) ctSamples, this, null);
}
```

Figure 149: The Routine Dictating the Structure of the Sample Space Resulting from a Combination.

[65] Note that while it would be desirable to maintain a more abstract framework in which variables other than the sample width of a dimension would be considered in deciding which dimension to downsample and where the user would specify an abstract cost threshold rather than a maximum sample count, the system currently does not support such an abstract approach.

The final bit of functionality to be presented in the handling of value combination is the routine that populates the elements of the output sample space with the combined samples, the remainder of the analogue to Figure 139. As seen in Figure 150, this task is accomplished by iterating through each coordinate in the output space (as determined by the routine *CreateCombinedSpace* shown in Figure 149), and combining the appropriate pair of samples (one from each of the two operands) which contribute to that output coordinate. In particular, recall that the output sample space contains a set of dimensions that includes all of the dimensions in either operand. Recall as well that by the rules of downsampling, any coordinate position along a given dimension in the output space must also be present in whatever operand value that dimension also occurs (this is enforced by the call to the least common multiple operator *LCMPowerOfTwo* present in Figure 149). In order to determine the coordinate within an operand that contributes to a coordinate in the output space, the system needs only to "project" the output coordinate into the sample space of the operand.

```
    private voidCoreComputeCombinedSampleSpaceFromDimensions(IntBinOp binOp)
    {
        SampleSpaceCoords coords = new SampleSpaceCoords();
        coords.Create(this.m_dimensions);
        //' ok, here we need to

    //'   for each of all of the possible pairs of coordinates in the exhaustive HybridDimension
s

    //'       iterate through each of the compatible tuples of coordinates in the sparse HybridD
imensions
        do

    this.SetSampleForCoordinates(coords, binOp.CombineSamplesForResultCoordinate(coords));
        while (!(coords.Increment()));
    }
```

Figure 150: Routine Used to Populate a Sample Space Resulting from a Value Combination by Combining Samples from the Sample Space of Each Operand.

This section has examined the infrastructure used to combine two sample space values using a binary operator. The process spans many levels of the abstraction hierarchy, and requires considerable machinery in order to determine the structure and propagate the elements of the output sample space.

11.4.5 Epistemic Sharpening from Dynamic Splits and the Greater Lower Bound Operator

Recall from Chapter 8 the fact that even when the analysis is unable to determine the truth status of a predicate directing run-time flow through a conditional, the truth status of the run-time predicate may be definitively known within many of the branches of the conditional. We can think of the analysis knowledge within the "true" branch of the conditional as the greatest lower bound of the analysis knowledge upon reaching the construct with a nearly identical abstract state in which the predicate is known to be true. Reflecting this fact, the compiler described in Chapter 7 ensured that at those control flow split points where

the system performs a dynamic split, (non-escaped) variables have their *GLBWithTruePredicate* method invoked, while the system calls the *DynamicSplit* operator of the current state domain. Earlier approximating domains had no general means of taking advantage of these mechanisms so as to "sharpen" the knowledge of an arbitrary value given information on the truth or falsity of a predicate. In particular, within such domains there was no way of establishing which potential values of a approximating domain value could arise in the same execution contexts as (i.e. are consistent with) a particular value (true or false) of a predicate. As a result, it is not in general possible to narrow the possible range of concrete values taken on by the approximating domain value given knowledge of the predicate truth or falsity.

By contrast, by virtue of disaggregating all non-uniform values by execution context, the sample space domain provides us with exactly this "value consistency" information. When the evaluation of the predicate yields a sample space value, each coordinate within that space will be associated with a value drawn from the three-level boolean domain. Only certain subsets of the execution paths can reach each possible path through the conditional. For example, only the execution contexts in which the predicate is associated either with a "true" or $\top$ value can reach the consequent branch of the conditional. By contrast, only those execution contexts in which the predicate has a "false" or $\top$ value will execute the alternate branch. (See Figure 151 for a depiction of this process.)

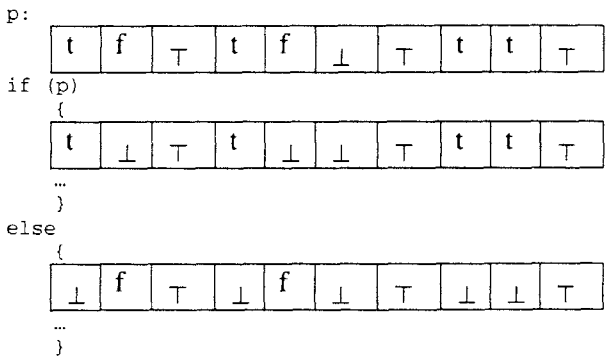

Figure 151: Selective Execution of Conditional Branches According to Value of the Predicate

Given the availability of a mechanism that allows for "sharpening" the knowledge of variables, the issue arises as to the best way to accomplish that sharpening. Two avenues are given below, each of which is straightforward to implement:

- **Per-Value Filtering.** One manner in which to sharpen values is to "filter" each value upon entry to a particular branch of the conditional. The interfaces specified in Chapter 5 and Chapter 6 provide the

means for carrying this out. The analysis framework invokes the *GLBWithTruePredicate* method automatically for (non-escaped) variables. Within a system using this technique, the result of invoking this method on a sample space value should be a "filtered" version of that value in which all execution contexts in which the predicate is associated with a false value are associated with a $\perp$ value. While it is in principle possible for the state implementation to iterate through stored locations and filter all values held internally, the cost of this would seem too high to realistically consider. This is particularly so on account of the fact that all such changes would have to be recorded in the "difference lists" characterizing one state from another (see Chapter 12).

- **Context-Based Filtering.** A more general strategy is to selectively filter values during *operations*, based on the execution context under which they are combined. The motivating realization here is that the same execution context "mask" is to be applied uniformly across all values within a particular branch of a conditional, and that it is too expensive to pre-apply this mask to *all* approximated values. Rather than pre-applying the mask to certain values up-front and then executing the code within the particular branch, this approach keeps track of the current execution context mask at each state, and imposes the mask whenever value combination takes place.

  This approach has the advantage of applying to *all* values and requiring no up-front masking effort, but is not without complications to the existing scheme. Switching to another state (e.g. when suspending one state and restoring another) requires additional recognition of the execution context masks in effect in the new state. Because the execution mask is *contextual*, a value cannot be completely interpreted without the knowledge of the associated execution context mask (and thus the state in which it is stored). Thus, taking the least upper bound of values drawn from two different states requires additional reasoning about the associated value masks. This is not an onerous requirement, but it does require some additional implementation effort.

The sole sample implementation of a approximating state in TACHYON does not currently take advantage of opportunities for epistemic sharpening, although extension of the system to make use of the per-value filtering would be straightforward and require only relatively minor effort.

11.4.6 Conclusion

The discussion above has offered a whirlwind tour of the sample space representation, outlining some of the strengths and weaknesses associated with this abstraction, and briefly surveying its implementation. This section attempts to summarize the more common and important tradeoffs that must be weighed when considering the use of the sample space representation, according to the same criteria that were used to

examine existing representational schemes in Appendix 2. For ease of reference, Table 33 includes the earlier table, but adds a final column to show where the two representations introduced in this chapter fit in.

	3-Level	Set-Based	Interval	Predicate -Based	Sample Space	Multi-Partition
Representational Breadth						
Compactness						
Cost of Value Operations						
Abstract Domain Height						
Epistemic Transparency				Depends		
Closure under Operation				Depends		
Scalable Analysis Precision						
Sensitivity to Frequencies	N/A					
Capacity to Maintain Contextual Dependency	N/A					

Table 33: Summary of Tradeoffs Between Alternative Value Representations.

11.4.6.1 Characteristics of The Sample Space Representation

The sample space representation is well suited to the representation of most types of program values, but falls short in the important case of representing pointers or references (an area discussed in more detail in Section 11.5.) Certain data types (particularly bytes or integers) will allow for exhaustive analyses (the "Holy Grail" of analysis in which the patterns of execution over the entire sample space are fully investigated) under certain conditions. This is much rarer for floating point values, whose use frequently indicates the presence of a *continuous* sample space whose exhaustive discrete approximation cannot be feasibly formulated. Nonetheless, the value representation offers a means to very precise characterization of computation on all other data types.

Capturing inter-value dependencies requires the disaggregation of values according to execution context, which – unlike any other value representation scheme -- necessitates the maintenance of large representations even for values that are relatively precisely known. This is ameliorated in two ways. First, the system uses a special representation for program values that are known to be uniformly associated with a

particular concrete value across *all* different values of a certain sample space dimension. Secondly, the sample space approximating value domain makes use of a downsampling methodology that provides an understanding of the statistical properties of the sample space without the need to exhaustively maintain all of that (abbreviated) sample space. Nonetheless, in the presence of all but the smallest domains, the space requirements of the sample space representation tend to be very large.

While value combination within the sample space representation will frequently be as simple as straightforward vector combinations, the cost of the individual value combinations under the sample space representation remains relatively high compared to analogous costs in most other representations. The two strategies that were discussed above to reduce the average size of a representation can also help to lower the cost of performing value operations. While the operations are expensive, the availability of inter-value dependency information does allows them to avoid the *combinatorial* costs that plague operations within the set representation (see Appendix 2) and the Disjoint Interval approximating value domain presented earlier in this chapter. As a result, the sample space representation scales far more effectively in contexts where high precision is desired but little can be deduced.

One of the strongest advantages of the sample space approximating value domain lies in the fact that while it can offer high accuracy, its lattice is very shallow. The height of the lattice is only two, allowing the domain that fixed point iterations undertaken with this representation will converge as rapidly as they do under the efficient 3-level domain discussed in Appendix 2. This permits better performance by algorithms despite the very significant cost of individual operations. The advantages in this area are particularly pronounced compared to the interval and set representations discussed in Appendix 2, and compared to the Disjoint Intervals approximating value representation presented in Section 11.3.

Like the other representations surveyed in Appendix 2, the sample space representation has an very simple and intuitive foundation. Unfortunately, the various mechanisms that play a critical role in reducing the cost of that abstraction also serve to obscure the interpretation of particular values. (This is particularly true of the abbreviated sample space and the downsampled sample space discussed in Sections 11.4.2.2 and 11.4.2.3 respectively.) Table 33 thus rates the transparency of the representation as rather low.

Unlike the interval representation (and certain symbolic representations), the sample space representation is closed under any possible combination of values.

From the start of this chapter, the emphasis has been on epistemic scalability. It should thus come as no surprise that scalability is one of the areas in which the sample space domain is strongest. This scaling can occur at two levels of granularity.

- At the coarsest level, the analysis designer can make use of a larger or smaller sample space for modeling the domain of interest. The size of the sample space will directly influence the time required for manipulation and computation of values, but will also affect the extent to which such values will accurately capture information regarding the entire set of possible execution contexts of interest.

- At a finer level, a threshold can be set that limits the maximum size of the value representation. If a value representation is produced that exceeds this size, it compressed in a lossy way and approximated by a "downsampled" version of the same value. The downsampling can be increased to provide for further compression.

The capacity to downsample values provides a very precise way of adjusting value precision. In contrast to the set-based representation, this can be done without adjusting the height of the abstract domain and in a manner that can be used for both precisely known and imprecisely known values. (Recall that "precision" is a characteristic of the *event* space, rather than the sample space – the extent of the sample space disaggregation is independent of the values in the sample space.[66]) This may allow the sample space representation to share with the 3-level representation the potential for use in algorithms requiring convergence time superlinear in the height of the lattice.

A major shortcoming of the sample space approximating domain lies in the fact that its precision scalability comes only at the cost of the ability to perform conservative analysis. In certain contexts where *understanding* of a program's dynamic operation is being sought, probabilistic analysis is acceptable, and can offer many of the advantages of exhaustive analysis without the high (and frequently entirely impractical) cost associated with such analysis. In such cases, the flexibility of the downsampling mechanism can allow the user to achieve a wide range of analysis precision/cost tradeoffs. In most contexts of traditional program analysis, however, performing a conservative analysis is essential to the correct operation of the collecting domains. In such cases, the sample space approximating domain carries a typically impractical cost, and offers the user no ability to adjust the cost of analysis other than by examining a smaller set of possible execution contexts.

[66] Note that this is not strictly true in a system such as that implemented, which has special representations for values which are *uniform* in the event space.

As suggested by the final rows of Table 33, the sample space representation offers considerably richer precision than do other approximating value domains. The first major advantage of the sample space representation relative to other representations lies in its ability to capture information regarding the *frequency* with which a program quantity takes on different concrete values. This information is maintained both implicitly by virtue of the representation's disaggregation of a program value across different execution contexts, and in the probabilities that are associated with each sample space element. Within a downsampled representation, the observed frequencies within the value representation will not be precisely the same as those within the entire sample space for that value, but will exhibit a distribution statistically similar to that of the entire value. Such frequency information can provide insights into program behavior, and permits analyses resources to be focused not just on those ranges of values in which the true runtime value of a program quantity *may* fall, but on those particular values in which such a value is *very likely* to fall. This variety of information is particularly valuable in contexts such as profiling, optimization, and in instances where a user is trying to understand the probabilistic behavior of the software system (e.g. in system dynamics simulations). This information is not available in any other representation of partially known values.

The second – and most important – advantage of the sample space representation is its capacity to represent the dependencies that exist between program values. This dependency information is implicit in the disaggregation of program values by execution context, and by the rule that dictates that a value specified for a particular execution context is *only* combined with values drawn from the *same* execution context. While the disaggregation of program values carries a heavy resource cost, the access to this additional information permits analyses using the representation to maintain exceptionally tight bounds on the values yielded by sequences of value combinations. This stands in contrast to the situation with the Disjoint Interval approximating value domain and the interval and set representations, where sequences of value operations frequently lead to serious degradation in precision that deleteriously affect the quality of subsequent analysis. (See Appendix 2 and Section 11.3).

11.4.6.2 Summary

In conclusion, the sample space representation is an expensive but very powerful way of maintaining information on *partially known* program values. Although it carries a space cost and per-operation performance overhead that can be extremely heavy, its shallow lattice allows for rapid convergence of fixed-point iteration.

Empirical experiments suggest that even relatively lightweight representations of partially known values that lack the capacity to recognize inter-value dependencies are unlikely to be worth the additional cost to

analysis performance that they impose. This shortcoming stems from the rapid loss of precision experienced by such representations when analyzing value combination. If a commitment is made to represent partially known program values within an analysis – and there are some good reasons for choosing to make such a commitment – then the choice should be made for a value representation that permits accurate modeling of such values throughout the program. Such modeling is only possible when using of a representation that observes and exploits the constraints that arises from value dependencies. For certain collecting domains, use of the sample space approximating domain can offer strong benefits.

Unfortunately, the costs associated with the sample space approximating domain make its use impractical in most analysis contexts. In particular, most analyses rely upon the approximating domain to maintain a conservative approximation to run-time behavior. In such cases, an *exhaustive* analysis is required, and within such an analysis the sample space approximating domain offers no capacity to soundly trade off analysis precision with computational costs. Within any situation in which the set of possible execution contexts is large, the overhead associated with an exhaustive analysis is prohibitive and it is infeasible to use this approximating domain.

In those limited situations in which probabilistic analysis is acceptable (typically cases in which user *understanding* is sought), the sample space representation of program values is capable of adjusting the precision with which program values are modeled in a fine-grained manner. These adjustments apply to values of any precision and can be accomplished without adjusting the height of the domain.

Although not without serious shortcomings, the sample space representation occupies a unique position among value representations by serving as a heavyweight representation for partially known values capable of reciprocating for its heavy use of computational resources.

11.5 Why Pointers and References are Different

The sections above noted the usefulness of the sample space representation for representing program values in certain contexts – for programs with small sample space, or for collecting domains in which probabilistic analysis is acceptable. The past two chapters have focused on approximating value domains for *integers*, and one issue not examined above is the possible applicability of the sample space approximating domain to the pointer domain.

This section argues that pointers are sufficiently different from scalar quantities that they will *not* benefit through analysis with the sample space representation. While an in-depth discussion of the issues related to pointer representation is reserved until Section 12.3, the brief list below attempts to highlight some of the

most important differences between pointers and other types that make pointers less suitable for characterization by the sample space representation. Further discussion on pointer domains can be found in Chapter 12.

- **No Combinatorial Loss of Accuracy.** Appendix 2 demonstrated the combinatorial loss of precision that is associated with combining two program values represented in the "set representation" used by [Osgood 1993]. As was argued in that section, any event space representation of program values will suffer from such a loss of accuracy during value combination. This loss steams from the fact that the contextual links between the possible concrete values taken on by program values being combined will not be recognized and observed.

  Pointer values differ from scalar values in that they more rarely (and in the case of references, never) undergo *combination* with another value. While scalar values frequently exist only as part of the flow of combinations (arithmetic, bitwise, etc.) that compute them, combination for pointers is vastly more limited. Even a pointer-oriented language such as C limits pointer combination to magnitude comparisons and a subtraction operator that returns the distance between the pointers; such operators are relatively rare. Given the greatly lessen role of value combination in the pointer domain, the strongest motivation for the use of the sample space domain does not apply.

- **Analysis Resource Constraints**. Pointers differ significantly from other built-in types in that they represent quantities that are actively created and destroyed by the program itself. For a given state in the program, we can ask a set of a well-defined questions for pointers, such as "Given the current state, is this pointer referencing a currently allocated region?" Maintaining information regarding memory layout and contents is important to high-precision modeling of pointer use. (See, for example, [Chase, Wegman et al. 1990]) While it is possible to represent pointer values using a sample space representation, the question immediately arises as to whether information concerning program state will also be maintained in a disaggregated manner (that is, with a separate simulated program state associated with each possible execution context.) Either option would appear to yield an unacceptable situation:

  - **Aggregated Memory Model.** A significant motivation for detailed pointer analysis would likely be based on its capacity to uncover allocation errors and bugs in memory management such as storage leaks and loose pointers. If state information is *not* maintained separately for each possible execution context, the value of maintaining disaggregated pointers will be sorely limited. (Certainly this is one of the most widespread uses of available analysis products.) Without the execution-

context specific disaggregation of memory state, it would be difficult indeed to detect such anomalies. For example, a memory block would have to be deallocated in *all* execution contexts before a loose pointer to that memory block could be recognized. Similarly, a piece of memory that is unreferenced in all execution contexts save one would not be recognized as a memory leak in the other contexts. The failure to maintain a memory model separately for each possible execution context eliminates most of the precision benefits that would normally obtain by modeling pointers in a disaggregated manner.

- **Disaggregated Memory Model.** If we choose to disaggregate the representation of the program state with respect to execution context, we will be faced at the least with a tremendous bookkeeping exercise, and very likely with an unrealistically large set of analysis computational resource demands for all but the very smallest sample spaces. Depending on the allocation behavior of the program, the representation of program state can require a tremendous amount of resources. (See Chapter 12) Accurately modeling the state in the presence of uncertainty and a guarantee of analysis termination requires considerable care and can significantly increase the resource demands needed to represent the state. The duplication of such a model for each possible execution context would seem to be simply impractical. While it is possible share aspects of the model of memory state between different execution contexts, accurately managing such shared components of the model entails considerable amounts of bookkeeping machinery and attendant complexity as well as exceptionally high space cost.

- **Importance of Pointers to Analysis Precision**. The program analysis literature has long recognized the critical importance of precise modeling of program pointers and references in analysis accuracy. [Chase, Wegman et al. 1990] [Horwitz, Pfeiffer et al. 1989] A substantial fraction of the analysis literature has been spent examining the issues in alias analysis and in devising algorithms for the analysis of pointer-rich code. Pointers and references are host to this dubious distinction by virtue of their promiscuous capacity to affect a wide set of other program values within the span of a single assignment. As a result, code involving pointers is frequently both far more difficult and more important to carefully analyze than is code manipulating scalars. In particular, as was argued in [Osgood 1993], an abstract domain of height two (comprised of $\{\perp$, concrete values, $\top\}$) is undesirably weak for pointer representation. Precise simulation of the potential effects pointer write requires the ability to drawn on a much richer set of information regarding the possible referents of the pointer. While the sample space approximating domain allows for collective characterization of a value as taking on a number of values, any given execution context of that value is associated with a domain

of only height two. This domain is too imprecise to capture much information on pointer referents for any particular domain, and will thus likely lead to loss of significant precision across *all* execution contexts during pointer writes.

- **Presence of a Natural Finite-Height Domain.** As is discussed further in Chapter 2, guaranteeing convergence in a finite time requires that the abstract state and pointer domain have finite height. Unlike most scalar types, analysis of the pointer domain can benefit from the availability of a natural and (in practice) compact finite-height domain for representing *sets* of pointer values. In order to ensure analysis termination, we can place regions allocated into equivalence classes associated with their points of allocation. Under this convention, the abstract analogue to a pointer to a particular concrete region will be an abstract pointer to the equivalence class associated with that region's point of allocation. Because the number of points of allocation is finite, the height of the state and pointer domain will be finite as well. Moreover, in practice we expect that the set of possible referents of a pointer will be rather small, because a given pointer is likely to mix with and refer to memory from rather few of such points of allocation. As a result, while the height of the powerdomain in theory would be equal to the cardinality of the set of all possible points of allocation of type *T*, the maximum length of any chain within the powerdomain that is climbed in practice is likely to be rather shorter.

The considerations above set pointers apart as significantly distinct from scalar values in their patterns of manipulation, their importance during analysis, their capacity for characterization using a set domain, and the extent of the resource commitment required to justify simulating pointers using a sample space representation. Each of these discussions weighs against the commitment of the resources necessary to make use of a sample space representation for the pointer domain.

Further discussion of pointer modeling in TACHYON that elaborates upon these ideas is found in Chapter 12.

11.6 Conclusion

The past two chapters have served as a broad-reaching overview of some sample approximating value domains. While the collecting domains examined in Chapter 6 collect user-defined information on value history, these domains direct the flow of execution in the analysis and are responsible for approximating run-time values. As the last chapter demonstrated, domains with simple structure and semantic rules are extremely easy to implement. For example, the toy even/odd domain and the sign approximating domain presented in Chapter 10 are quite compact, with the only complexity arising from software engineering

mechanisms used to maximize code reuse. By contrast, the scalable domains presented in this chapter are associated with a number of important advantages but require somewhat greater work to implement.

The multiple partition domain of Section 11.3 requires a modest amount of code to implement, but achieves scalable precision while avoiding some of the notable problems with other representational schemes (as noted in Appendix 2). Most importantly, this representation offers a compact and (statically) scalable means of capturing user-customizable information concerning the possible values of run-time quantities. While it is associated with a low domain height, it unfortunately suffers from combination time that can be quadratic in the number of partitions.

For truly in-depth analysis of run-time data patterns, the disjoint interval domain comes up short: It is incapable of capturing value dependency and frequency information, and because of the performance shortcomings cannot scale effectively to very high precision (with large numbers of partitions).

Section 11.4.1 introduced the sample space representation, which takes a novel approach to value representation by disaggregating values by *execution context* (a concept motivated and examined in much greater detail in Appendix 1). This disaggregation allows for the maintenance of value dependency and frequency information, which permits far greater precision when approximating the results of value combination. Unfortunately, the computational cost of this representation is very high: Despite techniques used to lower representation size, the space cost can be heavy. In addition, despite a very shallow domain, time cost for combination of domain elements is very weighty: While value combination does not suffer from the quadratic scaling that afflicts the multiple partition representation, every operation suffers the high *constant*[67] cost of application to many samples in the sample space. Nonetheless, the sample space representation looks likely to be valuable in certain specialized contexts, such as research into the asymptotic precision with which analysis can be performed, and use in analysis of relatively small systems (such as System Dynamics simulations).

Most importantly, these last two chapters has tried to demonstrate the wide variety of approximating value domain implementations that can be comfortably supported by the single, uniform set of interfaces presented in Chapter 5. We turn now to a discussion of an example approximating abstract state domain.

---

[67] Note that this is constant with respect to the level of knowledge concerning the value. The amount of work scales *linearly* with the count of execution contexts.

Chapter 12 Implementation of an Example Approximating State Domain

12.1 Introduction

The previous two chapters described sample implementations of approximating integral *value* domains. As the samples shown of Chapter 10 demonstrate, such domains can be very simple to implement. By contrast, while an approximating state domain can admit to trivial implementations, in order to achieve significant functionality it must implement much richer internal data structures.

Before beginning, it is worthwhile noting three requirements for modeling the abstract program state that set such modeling apart from what is needed in a standard semantics interpreter.

- **The need to simultaneously model several states.** Within the standard semantics, program execution proceeds according to a linear sequence of states. By contrast, in the *abstract* interpreter a particular state may be followed by more than one possible successor state. Regardless of whether the analysis proceeds in parallel or sequentially, evaluation in the context of such control-flow "splits" requires the ability to represent the contents of more than a single state over some period of time.

- **The need to model objects of unknown size.** During any point in the execution of a program in the standard semantics, all program quantities will be of some particular size. This is not necessarily true when we look at the execution of the program in the abstract semantics: Here, the size of a piece of memory allocated during program operation may not be known. (For example, an array may be dimensioned by parameters not available at analysis time.)

- **The need to model sets of allocated objects of unknown cardinality.** In the context of uncertainty concerning program execution context, the sequence of allocations and deallocations that take place during program operation cannot always be precisely simulated. For instance, the body of a loop that allocates objects may run an unknown number of times, yielding an uncertain number of allocations. A representation of program state following the loop must be capable of conservatively approximating the state that results from *any* number of memory allocations.

- **The capacity to identify the differences between any two arbitrary states.** A corollary to the ability to represent more than a single abstract state is the capacity to *combine* such states using operators such

as the least upper bound. Maintaining the ability to combine states in this way requires the rapid identification of points at which the states differ.

12.2 Approximating State Domain

As noted in Chapter 3 and Chapter 6, the approximating state domain carries with it a heavy cross-domain responsibility: It must approximate the standard semantics memory model for elements drawn from *all* value domains. In a more operational sense, the approximating domain must approximate the effects of all writes to memory locations and the values returned by all reads from memory locations. The abstract state interface discussed in Chapter 6 provides great flexibility as to the sophistication and efficiency as to the model of memory, and the requirement of correctness entails only that the abstract memory model represent an approximation to the standard semantics memory model, as discussed in Chapter 2.

12.2.1 Current State and the Memory Model Tree

As discussed in Chapter 6, the abstract state and domain interfaces adhere to a linearized model of execution in which there is a distinguished *current* state at all points, representing the existing point of execution. Although the use of a linearized model inhibits analysis parallelization, it promotes efficiency in a simpler and more obvious manner by allowing for the special treatment of the point of execution.

The sample approximating state domain presented in this section takes advantage of the distinguished character of the current state and the incremental nature of state modifications by representing the current state in a disaggregated manner, and expressing all other states relative to the current state.

In particular, as shown in Figure 152, all states are expressed as a directed "memory model tree". Each edge $A \rightarrow B$ in the tree is associated with a "difference list" (or "save list") which expresses the manner in which A differs from B.[68] Intuitively, a path from $A \rightarrow Current$ expresses how to reconstruct state A from the current state. Because all states are ultimately expressed relative to the current state, all (directed) paths in the tree terminate in the current state.

[68] Note that as discussed below, the "save list" or "difference list" is not in fact implemented as a list. But this historical name is reified in the code, and is retained here for the sake of consistency.

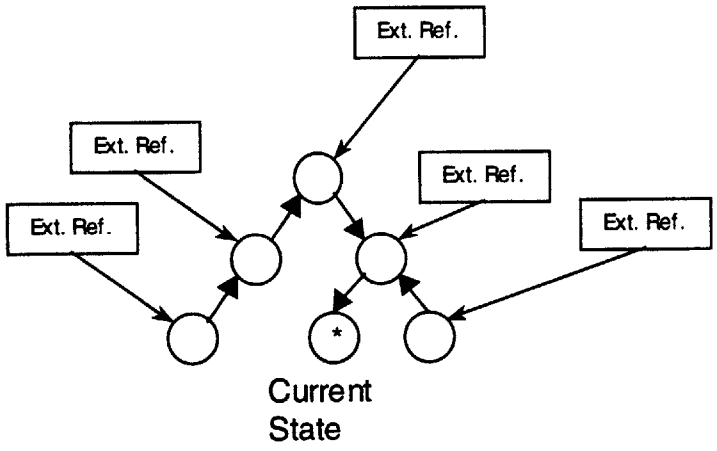

Figure 152: Depiction of the Memory Model Tree

The current state is the sole state modeled in a disaggregated manner: The system maintains information on the contents associated with different memory regions (each representing one or more concrete memory regions). Depending on user preferences, these models of memory locations may be disaggregated (in which case the contents of particular locations are separately approximated) or "collapsed" (where the locations within a region are approximated by a single "summary" value). A region contents model that starts as a disaggregated representation can be collapsed if the cost/benefit ratio of maintaining the disaggregated representation is judged to too high. (The judgement as to cost is made heuristically, on the basis of statistics concerning the precision with which information is read or written to the model.)

As discussed in the next section, the information maintained on the contents of locations is also held in close association with difference list entry information.

12.2.2 The Current Difference List

12.2.2.1 Introduction

The current state is itself changing at with each successive state modification (e.g. a write through a pointer value); all other states in the memory model tree represent static quantities that are simply expressed relative to the current state. Because the contents of the current state are changing but those of the other states are static, it is frequently necessary to update the information describing the static states in terms of the current state. In order to avoid the overhead of updating multiple difference lists for a given state modification, we maintain the invariant of having no more than one difference list enter the current state. If more than one path would join together at the current state node in the memory model tree, an additional node is created to

allow for the join, and a single difference list edge is extended from that node to the current node. (See Figure 153.)

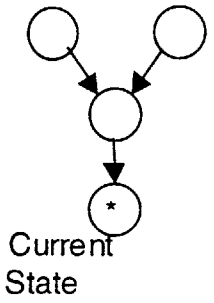

Figure 153: Avoiding the Need to Update Two Difference Lists by Use of a Single Difference List

Given a difference list *A→Current*, any given modification to a memory location *l* in the current state either requires an addition to the difference list *A→Current* (if no preexisting difference list entry exists for *l*), or no addition (if the contents of the location *l* in state A is already saved away in the difference list *A→Current*).

In this document, we will refer to the difference list associated with the single edge entering the current state node as the "current difference list". The implementation of difference lists is discussed further in Section 12.2.5.

12.2.2.2 Conclusion

This section has discussed the current difference list, and some of the motivations that have shaped its implementation within the TACHYON system.

12.2.3 Bottom Memory Model

In addition to representing states with information attached, implementing the state domain interfaces require us to represent a model of the $\perp$ element of the abstract state. This element is used frequently as an initialization value for variables that accumulate states via the application of the least upper bound operator while taking fixed-points.

Note that in the current analysis machinery (discussed in Chapter 8), the bottom memory model is never forced into becoming the *current* state. As describe that chapter, the machinery relies upon an exception (or

jump) mechanism pass control to the nearest possible branch point, which can then establish the appropriate abstract state replacement.

12.2.4 Modeling Allocated Areas

12.2.4.1 Introduction

A fundamental task of the abstract approximating state domain is to model the creation and use of memory regions. This is not as simple a task as it might seem – in order to guarantee convergence in a finite time, a potentially infinite series of standard semantics allocations must be folded into a finite series of abstract allocations. (See Chapter 6 and Chapter 8.) During analysis, it is in general undecidable whether the program being analyzed will allocate a bounded amount of space. Thus, at some point it is necessary to impose a state domain of finite-height.

Ensuring that analysis terminates in a *reasonable* amount of time may require considerable amounts of additional space-saving effort. This section describes the approach taken by TACHYON to each of these issues, and discusses the implementation infrastructure required to support this methodology.

As discussed in Chapter 2, Chapter 6 and Chapter 8, the use of a finite-height state domain is a sufficient condition for guaranteeing the convergence of program analysis. Achieving a finite-height state domain requires both the use of a finite height value domain and adherence to a memory model whose size must remain bounded throughout analysis. The fundamental tradeoff here is the traditional one that applies during analysis: Computational resources (time and space) vs. analysis precision. Rather than imposing a particular balance, TACHYON allows the user to explore a wide spectrum of possible tradeoffs – including the option of *not* enforcing analysis termination.

There are two components to a standard-semantics program that can grow without bounds throughout analysis: The program stack and heap. The modeling of each of these regions is discussed in the following sections.

12.2.4.2 Stack

As was discussed in Chapter 6, the responsibility for modeling the stack lies primarily outside the realm of the state domain. In particular, the standard semantics stack is modeled as a run-time stack in the abstract semantics[69], and (non-escaped) stack variables are independently maintained by the analysis. But the abstract state domains do receive notifications of stack-related activity that permit collecting domains to

[69] As will be described in Chapter 8, modeling of the stack in the presence of recursion makes the mapping less direct, due to the need to approximate stacks of arbitrary length by a single stack of bounded length.

accumulate information on various aspects of program behavior. The approximating state domain does not require information on stack activity and currently ignores these methods.

12.2.4.3 Heap

12.2.4.3.1 Goals

In comparison with stack allocation, heap allocation is more flexible, less structured, and more difficult to precisely approximate. A standard semantics program can allocate an arbitrarily large amount of space from an arbitrarily large number of allocation requests. As mentioned in the introduction, further complications arise because the size of a particular allocation may not be known during analysis, and the number of such allocations will typically not be statically available either.

The fundamental task at hand in approximating such allocations offering the ability to guarantee termination is the capacity to map a potentially infinite number of allocations onto a finite-sized set of equivalence classes. Ideally, the goal is to find a mapping which balances precision and storage requirements in the desired manner. That is, we seek a mapping that minimizes some user-specified cost function taking as parameters the amount of the useful information captured concerning the runtime execution of the program and the amount of space required by the representation. At a more intuitive level, it is desirable to find a "natural" mapping which "lumps together" different allocation occurrences in a way that doesn't lose much information on the contents of those allocated regions but also remains very compact.

12.2.4.3.2 A Popular mapping

A simple and popular strategy is to construct a mapping from the set of allocations onto a bounded-size range corresponding to the set of allocation points in this program. This strategy places into the same equivalence class allocations that take place at the same allocation construct in the program.

This strategy is attractive for a number of reasons. First, it is easy to define and implement. Secondly, it reflects the fact that frequently a given allocation point is used to create allocated regions with similar structure and use. Placing regions allocated by this region into the same equivalence class may allow for capturing these commonalties without undue loss of information. Thirdly, the set of allocation points within a program is frequently rather small, and the policy thus lends itself to a reasonably compact abstract representation.

A general disadvantage associated with the use of this mapping arises from its simplicity: In establishing the equivalence classes between allocation events and abstract regions, the mapping uses only information derivable from the program text (namely, the identity of the program point from which an allocation event

takes place). The failure to tap other sources of contextual information (for example, analysis information regarding the size of the region to be allocated, or knowledge concerning the broader execution context from which this call is occurring) prevents the strategy from taking advantage of other important patterns of similarity or difference among allocations events. Such patterns fall into two categories

- **Different use of same point.** For example, a particular user-defined allocation routine may be called from several different points within the program to allocate regions of very distinct size and structure. It seems arbitrary "clump together" all memory regions so allocated just on account of relying upon the same line of code to perform the allocation.

- **Similar use of different points.** Even if a set of different points are used to allocate regions of unknown size or internal structure, there is little benefit to be gained by modeling them using distinct abstract region approximations. Even if modeled independently, all such regions will likely be modeled as objects with unknown contents, and will derive little or no benefit from reification as distinct abstract regions.

12.2.4.3.3 The Sample Domain Approach

The sample domain uses a variant on the mapping strategy discussed above. In order to allow for the user to request more detailed analysis or full-fledged concrete execution when desired. Memory allocations take place entirely independently until a fixed-point analysis of the enclosing loop is begun. During fixed pointing, allocated regions are placed into equivalence classes in accordance with the program point at which they are allocated. During subsequent operation, the variant takes into account dynamic information regarding the size and composition of the allocated regions in order to discover additional opportunities for merging abstract models of allocated areas, and for collapsing erstwhile disaggregated representations of memory regions.

The subsequent sections describe the two fundamental abstractions used within the sample approximating state domain memory model and how those abstractions interact to consolidate the abstract representations. Figure 154 depicts the abstractions in use.

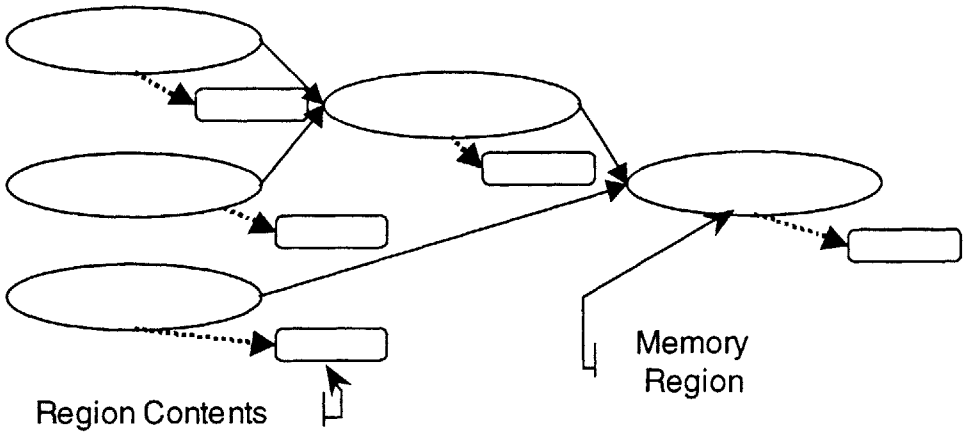

Figure 154: Depiction of a Relationship between Instances of the *MemoryRegion* and *RegionContents* Classes

12.2.4.3.3.1 MEMORY REGIONS

The highest-level allocation concept in the memory model is the *Memory Region*, which is reified in the *MemoryRegion* abstraction within the TACHYON code.

12.2.4.3.3.1.1 DESIGN GOALS

A particular memory region can represent one or more concrete allocated regions. Several priorities influenced the engineering of this abstraction. When designing memory regions, it was judged particularly important to create a representation that would allow for adaptive response to changing knowledge and constraints during analysis. Two types of adaptive response were viewed as especially desirable:

- **Capacity to Transparently Collapse Contents.** As was briefly remarked above, it is unnecessarily wasteful to maintain a high-quality, disaggregated representation of memory locations about whose contents very little is known. In such cases, we collapse the representation to a value consisting of the least upper bound of the contents of the entire memory region. It is desirable that this collapse occurs transparently, without the need to adjust any pre-existing references to the region contents.

- **Capacity to Transparently Unify Two or more Memory Regions into a Single Region.** Within the sample domain, initial equivalence class assignments for allocated regions are made purely on the basis of points of location. Further space savings are then achieved by "unifying" similar regions from

distinct allocation points into a single region. As for the collapsing, this unification should take place transparently, so references to either of the regions created prior to unification will remain valid following unification.

The capacity to collapse the representation was realized by designing the memory region so as to delegate memory contents modeling to another abstraction – the *RegionContents* abstraction discussed in the next section.

The desire to empower transparent memory region unification led to the use of a standard delegating union/find architecture, in which equivalence classes are represented as trees in which each edge represents the delegation of one memory model to another in the same unification equivalence class. In order to allow for trees that are as "bushy" as possible, unification always grafts the smaller tree onto the larger.

Figure 154 depicts a typical scenario illustrating the relationship between *RegionContents* and *MemoryRegions*.

12.2.4.3.3.1.2 TAXONOMY OF MEMORY REGIONS

There are two types of memory regions:

- **ConcreteRegions.** Each of these regions represents one or more regions allocated at runtime.

- ***-Region.** A particular *-Region represents an *arbitrary* number of possible runtime regions. These regions are produced by the allocation machinery in the context of fixed-pointing operations. Under such circumstances, the results of a given abstract allocation may approximate an unbounded number of concrete allocations, and the *-Region approximates the regions associated with each such possible allocation.

12.2.4.3.3.1.3 MEMORY REGION CONTENTS

Each *MemoryRegion* object is associated with several pieces of information:

- **A version number.** This is used to determine whether a particular difference list entry (see Section 12.2.5) refers to the existing region or to a former version of this region prior to a point at which it was unified together with the memory contents of another region to reduce space.

- **Delegation Information.** This information specifies another region with which this region shares representational space and to which this region should delegate operations.

- **Bookkeeping information.** The memory region maintains bookkeeping information for three purposes: To expedite determining whether region-level difference list entries are contained on the current difference list, to aid in determining the number of distinct regions pointed to by a pointer with many possible referents, and as a count of the nodes in the *MemoryRegion* tree beneath this node (used to enable an efficient union algorithm).

- **Contents Information.** The most widely used information within a Memory Region is that representing the *contents* of the region – the memory locations and their contents that are approximated by this region. As noted above, the contents are encapsulated in a Memory Contents data structure (see Section 12.2.4.3.3.2).

12.2.4.3.3.1.4 POINTER COMPARISON SEMANTICS

As will be discussed in Section 12.3, abstract pointers refer to a set of offsets in *MemoryRegions*. Semantic rules governing the equality test between two particular referents for an abstract pointer is shown in Table 34. The entries in the table indicate the return value for the equality operator. (In essence, indicating whether the values refer to the same *concrete* region -- i.e. are guaranteed to both point to a piece of memory allocated at the same point at run-time). The result of the comparison depends not only on whether the pointers point to the same *abstract* region, but also details concerning the type of region involved, and information concerning the region contents associated with that region.

Intuitively, the rules can be phrased as the following:

Suppose the two pointers refer to the same abstract region. Then if that is a concrete region with an *unshared RegionContents*, the pointers definitely point to the same concrete region. Otherwise, they may or may not point to the same concrete region.

If the two pointers refer to *different* abstract regions, then we know that there is no way we can *definitely* know whether they point to the same concrete region – but we might be able to guarantee that they definitely *do not* refer to the same region at runtime. In particular, if they do not share the same region contents, we know that they *cannot* refer to the same run-time region. Otherwise, they may or may not be able to do so – we have too little information to tell.

Pointers refer to same region?		Same	Different			
Kind of other Region		Same	ConcreteRegion		*Region	
Regions share RegionContents?		Yes	Yes	No	Yes	No
Region Type?	***ConcreteRegion***	*1 ref: True* *>1 ref:* T_{bool}	T_{bool}	*False*	T_{bool}	*False*
	****Region***	T_{bool}	T_{bool}	*False*	T_{bool}	*False*

Table 34: The Rules for Comparing Two Abstract Pointers.

12.2.4.3.3.2 REGION CONTENTS

The *RegionContents* abstraction encapsulates information concerning the locations and contents associated with an allocated region. Region contents are responsible for representing the disaggregated contents associated with the "current state" (see Section 12.2.1), and as such are the final target of all memory reads and writes. As suggested by Figure 154, each memory region is associated with a *MemoryContents* object that encapsulates the locations within that region. A particular *MemoryContents* object may be shared between one or more *MemoryRegion* objects. The region contents include within themselves the entries of the current difference list (see Section 12.2.1) that express the first step of how to recreate all other "suspended" states from this one. (For example, the current difference list would express how to recreate the conditional-entry state while processing the "true" branch of a conditional.)

12.2.4.3.3.2.1 TYPING ISSUES

An important assumption of the current TACHYON system is the ability to associate each element of a memory structure with a well-defined statically determinable type. The ability to perform such strong typing eliminates the need to deal with the extremely awkward task of creating abstract models of reads of (abstract) values from locations that were written using another type or that were composed of partial fragments of one or more abstract types.

While this thesis originally aspired to model untyped heap structures and such forms of value manipulation (which enjoy relatively frequent use in weakly typed languages such as C), it was decided that the mechanisms and bookkeeping needed to successfully model such behavior would obscure the primary goals of the thesis. It is unclear to the author whether truly high-precision analysis of the type performed by TACHYON is realistically achievable on sections of code that exhibit of such behavior.

12.2.4.3.3.2.2 MODELING RECORD TYPES

For simplicity of modeling, all *RegionContents* are currently homogeneously typed – the system currently has no support for declaration or modeling of record types, although it would not be difficult to add. The most important change necessary to model records would be the ability to represent *multiple* "summary" values in a collapsed region (see below), either with one for each type or, with one for pointer types and one for other element types. (The motivation for the segregation of the summary regions of different values is the fact that taking the least upper bound of data types that use different abstractions is not currently well defined.) It is unclear whether it is appropriate to make use of a collapse routine for record types.

12.2.4.3.3.2.3 DISAGGREGATED AND COLLAPSED CONTENTS

12.2.4.3.3.2.3.1 Motivation For Collapse

There are two types of Region Contents models: Disaggregated models in which each element is individually approximated, and collapsed models in which all elements are approximated by a single "summary" value. Transitions between the disaggregated and collapsed forms occur in one direction only, and are currently irreversible.

A collapse is desirable under any of several conditions:

- **The elements contain too little information**: Looking at the issue from an information-theoretic perspective, this applies in two familiar cases:

  - **The elements are poorly known.** In this case, the system's knowledge concerning element values is too imprecise to make it worthwhile to maintain the information in a disaggregated state.

  - **The information can be stored more compactly.** In this case, the current location contents are too uniform to justify a disaggregated representation.

  It is important to note that the concept of "information" in the heuristics discussed above should properly include both elements of the approximating and other domains. (For example, in determining whether too little is known about the elements, it is important to examine the positions of the elements in both the approximating and non-approximating domains.)

- **The information is of too little importance to be worth maintaining.** Judging the "importance" of certain analysis information to the analysis is difficult. One heuristic to use would be the frequency with which the information has been used – if the information in a certain region were accessed

infrequently by the analysis thus far, it may be a hint that they will offer little value as the analysis proceeds. Lacking any knowledge about future analysis patterns, any such estimate will be risky.

- **The information is accessed at too coarse a granularity to make disaggregated access desirable.** If an array is modeled in a very detailed manner but is always read through imprecisely known indices, the high fidelity of the contents offers little benefit. Similarly, writes to imprecisely known array indices will quickly destroy much of the information that is present even in the most disaggregated model of a memory region.

Although it is possible to have a system monitor for all of these conditions, currently TACHYON only monitors for cases in which the final condition obtains. In particular, the system counts the number of reads and writes to the region conducted through unknown offsets; if either of these reach a user-specific threshold, the array is collapsed. With this infrastructure in place, any of the other conditions could be implemented with very little effort.[70]

12.2.4.3.3.2.3.2 Motivations for the use of a summary value

At first appearances, it may seem unnecessarily wasteful to represent a collapsed region by any value at all: For a sufficiently large array, it might be expected that the summary value would almost certainly be at or near the top of the lattice.

There are at least two good reasons to maintain a summary value:

- **Pointer types require particularly delicate modeling.** Using the lattice established in the sample pointer domain, it is unlikely that the least upper bound of an entire array will approach the top value for the domain. Approximation of each element of a collapsed pointer array by the top value would thus yield a strong decline in precision in return for a negligible space and time savings.

- **The lattices associated with a user's domain could be associated with very deep lattices.** Assuming that the least upper bound of an array will be close to the $\top$ value is thus somewhat risky. Note that the pointer domain is just one particularly notable case of this general caveat.

- **The array elements may be unusually uniform in one or more domains.** Maintaining a summary value will capture this fact, but the knowledge would be lost if the region were eliminated.

[70] Note, however, that the computational and/or bookkeeping cost for some of these conditions will be considerably larger than for the current metric.

12.2.4.4 Summary

The sections above have discussed the fundamental abstractions used in modeling the program heap. Both in order to guarantee termination and to allow for lighter-weight analysis, the representations are capable of mapping the potentially boundless set of run-time heap regions onto a fixed set of abstract heap regions. This mapping takes place only during fixed-point iteration, giving the user the option of full-bodied (and potentially non-terminating) abstract execution where desired. The abstract state abstractions presented here offer are distinguished in that they are *adaptive*, dynamically recognizing and taking advantage of opportunities to save space and analysis time in situations where the precision advantages of a high-fidelity representation are small.

The abstractions described above offer two means of trading precision for lower computational resource requirements. First, the *RegionContents* abstraction can abandon a fully disaggregated representation and collapse itself down to a single "summary" value in cases where the disaggregated representation appears unlikely to offer many benefits. Secondly, previously distinct *MemoryRegions* can be "unified" together in order to share a *RegionContents* representation for the associated locations.

While the current strategy implemented within TACHYON is not as aggressive as it could be in pursuing such resource-saving changes, the large majority of the machinery is currently in place to allow for the straightforward implementation of stronger techniques.

12.2.5 Difference List implementations

Section 12.2.2 provided a brief survey of some of the issues that arise in the context of modeling difference lists, and discussed in particular the concept of a current difference list. This section examines some of the implementation issues that arise in the context of difference lists.

12.2.5.1 Current Difference List Implementation

12.2.5.1.1 Motivation for Differential Treatment of Diff List Entries

Just as states can be divided into "non-current" and "current" states, difference lists can be classified as "non-current" and "current" difference lists. The current difference list the distinguished difference list associated with the current state, and – like the current state -- is implemented in a distinctive way that fosters efficient modification.

In particular, while the per-location entries for non-current difference lists are reified in distinct data structures, the entries associated with the current difference list are not separately represented. Instead, they

are incorporated into the data structures that represent the current state of memory. While this is accompanied by a considerable space cost, it offers a number of benefits:

- **Rapid Creation of Entries.** The maintenance of difference list entries in association with the locations to which they pertain allows for very rapid checking if an existing entry already exists for a particular location, and minimizes the amount of bookkeeping machinery that must be invoked to create the entry.

- **Locality of Reference.** Storing information concerning the difference list status of a given location alongside the location lends spatially locality to already temporarily close references and eliminates the time and space overhead associated with additional bookkeeping to relate the location to its associated difference list entry.

- **Batched creation of entries.** In essence, maintaining difference list entries with locations allows for the "batching" of external difference list creation. Rather than creating an external difference list on the fly as the state is being modified, we only externalize the difference list once a new current state and current difference list is selected – and once the contents of the difference list are definitively known and no longer subject to modification. This offline creation of the external difference list allows a more judicious selection of external difference list structure than is possible with online creation. For example, consider an approach in which an external difference list is created on the fly. Suppose the approximating state domain must handle three writes to a contiguous block of three locations in the order l, $l+2$, $l+1$. At the point at which the system is creating the different list entry for the location $l+1$, two independent existing entries already exist for l and $l+2$. Given that the locations are contiguous, it would be desirable to maintain all three locations in a more compact manner. Unfortunately, due to the online strategy employed, this would require deallocation of the earlier entries, and recreation of a new entry, not to mention the appropriate bookkeeping necessary to recognize and exploit the opportunity for the consolidation. By contrast, the "batched" external difference list creation delays the choice of the external difference list entries until all locations which must be saved are known. This permits a contiguous external difference list entry representation to be selected at the point when the external difference list is first created.

- **Merging External Entries with Current State.** Any system that maintains an external difference list representation for the current state will face the need to merge that representation with difference lists associated with other edges in the memory model tree. (For example, while taking the least upper bound of the approximating domains of two abstract states). Once again, the amount of work required

in order to perform this on a current difference list that is stored in association with the location contents information proves to be less than that required to reconcile two external representations.

- **Simpler Consistency Maintenance.** As will be discussed further in Sections 12.2.4.3.3.2 and 12.2.4, the models of memory contents maintained by the system are "adaptive representations", in the sense that they can be modified after creation based on patterns of usage and space considerations. For example, a large number of unknown reads or writes to a large array might suggest that an erstwhile disaggregated model of the array would offer little benefit. Based on such considerations, the array could be "collapsed" to a much more compact representation. Similarly, two previously distinct memory regions with similar size or contents might be "unified" together so as to maintain joint storage. An external difference list would require adjustment to account for the fact that the fine granularity of its description of longer contents may no longer be appropriate. On the other hand, if the current memory model is stored in conjunction with the information concerning memory model contents, no additional work is needed.[71]

12.2.5.1.2 Overall Structure

The current difference list consists of a linked list of references to memory regions, as encapsulated in the *CurrentDiffListEntry* abstraction. Each of these regions internally encloses (potentially null) difference list entries on each of the locations, including information specifying the contents for that location in the previous state. (Recall that the purpose of the difference list entries is to enable to recreation of that previous state from the current state). Adding additional information to and reversing the direction of the difference list can both be rapidly performed by the memory region itself.

12.2.5.1.3 Externalization and its Reverse

When the current state shifts to a previously saved-away state, the existing current difference list must be externalized, and a new difference list must be instantiated.

The externalization process is simple: Each region specified in the current difference list is requested to externalize its contents. If the region is collapsed, it merely creates a collapsed difference list entry for the entire region. For disaggregated regions, the region will in general create a number of intraregion difference list entries, with each entry corresponding to a contiguous sequence of locations within the entry for which

[71] Note, however, that the system *will* have to be able to recognize cases in which existing difference lists between saved-away states have "obsolete" entries that refer to memory regions that have since been updated. The "version" identifier maintained with each MemoryRegion and external difference list entry serves exactly this purpose.

saved-away information is present. (Note that if this memory region is at some point collapsed, these entries will be needlessly disaggregated, and can be converted to collapsed form.)

The instantiation process is equally simple: Difference list entries are simply read and instantiated by the *SavedValues* abstraction that encapsulates the per-location saved information within each disaggregated *MemoryModel*.

12.2.5.2 Non-Current Difference List

Non-current difference lists are static, and will not benefit to efficient random access of per-location data. The emphasis in the implementation for such abstractions is instead on compact representation of the saved information. External difference lists are composed of a linked list[72], each of whose links is of type *RegionDiffListEntry*. Each link maintains saved contents information for locations within a particular region and is marked with a version identifier. This identifier helps detect whether the recorded entries relate to the region at the current level of aggregation, or whether the difference list entry was created before the most recent aggregation on the region was created.

Each *RegionDiffListEntry* includes a sequence of elements of type *IntraRegionDiffListEntry*, each of which maintains information on a contiguous sequence of elements of the associated *MemoryRegion*. There are three types of such entries:

- *Collapsed entries* maintain a single value approximation for all elements of the region contents interval.
- *Interval entries* maintain a single value approximation for each element of an interval.
- *Single element entries* maintain a single value approximation for an element at a particular index of the underlying *RegionContents*.

Figure 155 depicts the structure of an example non-current difference list.

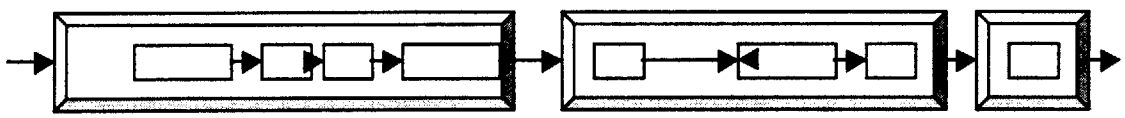

Figure 155: Depiction of the Instances of *RegionDiffListEntry* and *IntraregionDiffListEntry* in a Non-Current Difference List.

[72] Given the static character of non-current difference lists, this should probably be implemented as an array.

12.2.6 Interface Method Implementations

In the light of the discussion above, this section briefly surveys the implementations of different state interface methods.

12.2.6.1 State Join

Figure 165 shows the implementations of the least upper bound operator for the sample approximating state domain. This routine occurs at state joins following conditionals and during loop execution, and plays a critical role throughout the execution of the program. Recall that all abstract states are expressed relative to the current abstract state. The routine performs the least upper bound by walking the "difference lists" separating each abstract state from the current state. In particular, the routine traverses the "memory model tree" discussed in Section 12.2.1 along the (directed) path from the non-current state to the current state. Each difference list along the edges of this path can be thought of incrementally describing the ways in which the non-current state differs from the current state. For each difference list, the system acts upon entries for locations that have not been previously seen on earlier edges. In particular, if a given location has not previously been spotted on the path from the non-current state, the saved-away value associated with this location must indicate the value for the location in the context of the non-current state. The least upper bound is then taken between this saved-away value and the current value, and the result stored to the current abstract state (recall that the least upper bound is occurring here *to* the abstract state)

```
    public void LUBToCurrentState(_IStateDomainInfo dmnCurrent, int srcLoc)
    {
        // ignore return value
        this.FLUBToCurrentState((GuidingStateDomain) dmnCurrent);
        return;
    }

    private boolean FLUBToCurrentState(GuidingStateDomain  dmnCurrentNarrowed)
    {

    boolean fChanged = dmnCurrentNarrowed.m_node.LUBToCurrentState(s_passMMTreeNodeByRefHack, th
is.m_node);

        if (s_passMMTreeNodeByRefHack[0] != dmnCurrentNarrowed.m_node)
        {
            dmnCurrentNarrowed.m_node.DecrementCtExternalReferences();
            dmnCurrentNarrowed.m_node = s_passMMTreeNodeByRefHack[0];
            dmnCurrentNarrowed.m_node.IncrementCtExternalReferences();
        }

        s_MRGuidingStateBooleanReturnValue = fChanged;
        return fChanged;
    }
    // note that here (unlike the routines that call this from AbstractState), this = curr

    // the semantics of this is that we return a boolean indicating if
    //  LUB(other, current) > other   in the lattice (i.e. if either
    //  other < current OR neither (current < other) nor (other  < current)

    // curr <- LUB(other, curr).   fChanged = (LUB(other, curr) > other)
    // again, note that this = curr
```

```
    boolean LUBToCurrentState(MMTreeNode nodeOut[], MMTreeNode n)  // curr <- LUB(curr,this)
    {
        // this is a simple case: we are keeping the current state at the same position

        // this is the current state
        //  n guaranteed non-null

    // we have to see if the current node is shared.  if it is, we need to create a new subnode

        // we must guarantee that there is only one incoming edge into the current node
        //  moreover, we need a NEW CURRENT DIFF LIST so that we can process all of the entries
        //  of the current one without then having to process that's we've just lubbed.

    MMTreeNode nodeNewCurrent = this.GetWriteDestinationAfterPossiblyChangeStateToEnsureUniqueIn
comingDiffList(false);
        // locations on the diff list are already marked as such
//      nodeNewCurrent.SetLocationsOnIncomingEdgeDiffListsAsAlreadyArchived(s_idTraversal++);

        s_idTraversal++;
    // ok, we are starting a new traversal here.  bump up the the traversal count.
        boolean fChanged = false;

        // retrace the steps from the specified node to the current node
        while (n != nodeNewCurrent)
        {
            // for each location listed in the diff list,
            //
    if the location has already been recorded, then we've already lubbed the current value and t
he value that obtains at n =>  do nothing
            //
    if the location has already not been recorded, then dl contains the value that obtains at lo
cation l at n =>
            //  check if this location is already recorded on all incoming edges for this.
            //      if so, there's no need to save away its location --
 none of the memory models that depend on this one will be affected
            //
    otherwise, we need to place a location for this entry on the dl of each incoming edge.
            //          the value associated with this entry is the pre-
existing value (b/c there was no entry already, we know
            //
    that the node on the other side of the incoming edge must have the same value for this locat
ion as does this preexisting state.
            //
    just LUB the contents of the found entry and the current value at that location, and place i
t in memory

            MMTreeEdge edge = n.m_rgEdges[n.m_iOutgoingEdge];
            DiffList dl = edge.m_diffList;


    // the deal here is that since we are CHANGING the current state, in general we will need to
 add to the
            //  "diff lists" that separate the nearest neighbor MMNodes from the current state


    // note that a tricky case arises for those diff list entries which need to be added which a
re associated
            //
     with areas neither disjoint from nor identical to areas already in the current diff list.
            //  whenever we have such partial-
overlap regions, we advance the MMNode for teh current state so
            //
    as to form a new DiffList.  (note that this is why we can't deal as easily with LUBbing TO s
aved-away states.

            fChanged |= dl.PerformLUBsToMemoryAndPossiblyArchiveUnmarkedLocations(s_idTraversal);
            n = edge.m_nodeTo;
        }

        nodeOut[0] = nodeNewCurrent;
        return(fChanged);
```

```
}
```

Figure 156: Handling of the Least Upper Bound Operator in the Sample Approximating State Domain.

12.2.6.2 Splitting

As described in Chapter 6, the dynamic split operator is used to create a branch state associated with control-flow resulting from a predicate whose truth value cannot be determined at the point of analysis. While the interface method *DynamicSplit* is passed an integer value specifying the predicate value, the approximating state domain does not currently make use of this value. Instead, the current state is simply "cloned" in method *SuspendState* and returned. Note that in truth, the "cloning" does not do any substantial copying of a states contents, and the resulting "cloned" state will still refer to the current node in the "memory model tree". The effective copying of the current state is carried out, however, by indicating the existence of an external reference to the current node in the memory model tree through incrementing the reference count for this node. The existence of an external reference to this (erstwhile current) node will ensure that future modifications to the current state will leave the state at this node unchanged.

```
public  _IStateDomainInfo DynamicSplit(_IIntDomainInfo vCalculated)
{
    s_MRGuidingStateStateReturnValue = this.SuspendState();
    return(s_MRGuidingStateStateReturnValue);
}

public _IStateDomainInfo SuspendState()
{
    m_node.IncrementCtExternalReferences();
    s_MRGuidingStateStateReturnValue = GuidingStateDomain.this.Clone();
    return(s_MRGuidingStateStateReturnValue);
}
```

Figure 157: Routines to Handle State Splits and Suspends in the Sample Approximating State Domain.

12.2.6.3 Memory Access

Figure 158 shows the approximating state domain interface methods that handle pointer reads and writes. For the most part, the routines simply delegate the work to the appropriate methods of the approximating pointer domain (recall from Chapter 5 that the approximating pointer domain is responsible for reading and writing values across *all* domains). Section 12.3.1.3.1 illustrates the implementation of the delegated methods by the sample approximating pointer domain.

The handling of write requests necessitates an additional bit of work. In particular, if a "snapshot" of the current state has just been taken (e.g. in a join) and a write request must be performed, a new current state must be created, so as to avoid altering the contents of the snapshot. The *WriteInt* routine calls off to an appropriate routine to handle this case prior to delegation to the pointer value.

```
        public _IAbstractIntValueReadInt(_IGuidingPtrDomainInfo loc, int srcLoc)
        {
            return(loc.ReadInt());
        }
        // note that this is called AFTER any domain-
specific filtering has been performed by each of the domains
        public void
    WriteInt(_IGuidingPtrDomainInfo loc, _IAbstractIntValue argRHSAfterAssignmentNotificationAnd
Copying, int srcLoc)
        {
            // if necessary, advance the current state to a new node
            //  NB: This will have bumped up the traversal id if we needed a new node


    MMTreeNode n = GuidingStateDomain.this.m_node.GetWriteDestinationAfterPossiblyChangeStateToE
nsureUniqueIncomingDiffList(false);

            if (!n.equals(m_node)) // if we do have to advance, do the bookkeeping
            {
                m_node.DecrementCtExternalReferences();
                n.IncrementCtExternalReferences();
                m_node = n;
            }

            // there is no reason to pass this write on to the nodes -- they just keep track of
            //   the relative state positions.

    // instead, the write is handled by the lvalue, and then by the region and its contents

            loc.WriteInt(argRHSAfterAssignmentNotificationAndCopying);
        }
```

Figure 158: Pointer Memory Access Routines for the Sample Approximating State Domain.

12.2.6.4 Allocation

The final fragment of implementation functionality we will examine for the sample approximating state domain is that associated with memory allocation. Figure 159 shows the routine handling a request to allocate an integer array. The functionality of this routine differs strongly between times when the analysis is performing a fixed point and when it is not involved in a fixed point approximation.

- If a fixed point is being sought, the routine returns a canonical approximating pointer domain value associated with the point of allocation (as given by the parameter *iSrcLoc*). As discussed in above, this allows the analysis to maintain a pointer lattice of finite-height, thus guaranteeing convergence of the fixed pointing within a finite time. The canonical *MemoryRegion* associated with the pointer returned by the allocation when fixed-pointing is a *-Region (see Section 12.2.4.3.3.1.2) – a region that represents an arbitrary number of concrete regions. The semantics of such regions is discussed in Section 12.2.4.3.3.1.4.

- If no fixed point is taking place, the system simply allocates and returns a reference to a *ConcreteRegion*, which approximates (at its time of creation) a single allocation in the standard semantics. Figure 160 shows the routine to which *AllocateIntArray* delegates in this case. The

allocation of a new *MemoryRegion* requires creation of an associated *RegionContents* structure (see Section 12.2.4.3.3.1.3). The type of *RegionContents* object that is allocated depends on the count of locations specified by the code: If there is a concrete number of elements specified, the system allocates a *RegionContents* of the appropriate size. In other cases (where the count of elements is imprecisely known or completely unknown), the system allocates a *collapsed RegionContents* object.[73]

```
    public _IPtrDomainInfo AllocateIntArray(_IIntDomainInfo ctElts, boolean fPerformingFixedPoin
t, int iSrcLoc)
        {
            RegionContents contentsTmpAlloc;
            _IGuidingIntDomainInfo i = (_IGuidingIntDomainInfo) ctElts;

            if (fPerformingFixedPoint)
            {
                Object keySrcLoc = new Integer(iSrcLoc);

    if (s_mapAllocationSrcLocToCanonicalFixedPointMemoryRegion.containsKey(keySrcLoc))

    return((_IPtrDomainInfo) s_mapAllocationSrcLocToCanonicalFixedPointMemoryRegion.get(keySrcLo
c));
                else
                {

    _IPtrDomainInfo vReturn = AllocateIntArrayInternalNoFixedPtCache(ctElts, fPerformingFixedPoi
nt, iSrcLoc);
                s_mapAllocationSrcLocToCanonicalFixedPointMemoryRegion.put(keySrcLoc, vReturn);
                return vReturn;
                }
            }
            else

    return AllocateIntArrayInternalNoFixedPtCache(ctElts, fPerformingFixedPoint, iSrcLoc);
        }
```

Figure 159: Implementation of Two Allocation Methods for the Approximating State Domain.

[73] As discussed in Section 12.2.4.3.3.1.3, the sample approximating state domain currently only supports completely disaggregated value and completely collapsed values, but the system provides a hook to allow for extension of this functionality by specifying the *range* of sizes that might be associated with the allocation request to the routine that creates the appropriate *RegionContents* object. This allows support for *RegionContents* objects of partially known size to be added transparently with respect to the code shown.

```
    private _IPtrDomainInfo AllocateIntArrayInternalNoFixedPtCache(_IIntDomainInfo ctElts, boole
an fPerformingFixedPoint, int iSrcLoc)
        {
            RegionContents contentsTmpAlloc;
            _IGuidingValueDomainInfo iGuidingGeneric = ((_IGuidingValueDomainInfo) ctElts);
            _IGuidingIntDomainInfo i = (_IGuidingIntDomainInfo) ctElts;

            if (iGuidingGeneric.FConcrete())
                contentsTmpAlloc = new IntArrayRegionContents(i.SSValue(), iSrcLoc);
            else

    contentsTmpAlloc = new IntCollapsedRegionContents(i.MinPossibleSSValue(),i.MaxPossibleSSValu
e(), iSrcLoc);


            return CreateFromRegionContents(contentsTmpAlloc,  fPerformingFixedPoint, iSrcLoc);
        }
```

Figure 160: Code to Handle Allocation of a New Memory Region in the Approximating State Domain.

12.3 Pointers

12.3.1.1 Introduction

This section discusses the sample approximating value domain used to represent pointers. Due to the high precision generally desired for pointers, the sample representation describes pointers as a *set* of possible referents. As was noted in Section 11.5, this representation is more amenable for pointers than for scalar values, due primarily to the greater domain height sought for pointers as well as the low relative frequency of pointer combination.

12.3.1.2 Implementation Overview

The GuidingPtrDomain class implements the approximating pointer domain. The key data structure within this abstraction is a sequence of *AtomicPtrReferent* objects, each of which represents a possibly referent of the pointer that consists of a *MemoryRegion* associated with precisely or imprecisely known offset. The *GuidingPtrDomain* abstraction maintains a canonical ordering of the *AtomicPtrReferent* sequence and observes the invariant that the *AtomicPtrReferent* elements are mutually disjoint. The *GuidingPtrDomain* abstraction is thus viewed as encapsulating a collection of disjoint potential referents for a pointer.

As remarked above, each *AtomicPtrReferent* is associated with a particular *MemoryRegion*, and an offset within that memory region. Within the current implementation, the offset may either be a precisely known (concrete) value or an unknown quantity. Note that while the *AtomicPtrReferent* does not currently support partially known indices within a region, such support would be relatively easy to add.

The representation of a pointer as a set of disjoint possible referents allows for relatively straightforward implementations of the various interface methods. In the spirit of the descriptions of integer approximating

value domains discussed in Chapter 10 and Chapter 11, the next several sections briefly examine the implementation of several categories of value.

Through an oversight in the existing implementation, currently no representation exists for the null pointer element. This would be extremely simple to add, by making *AtomicPtrReferent* an interface, and creating another *NullPtrReferent* implementation of that interface.

12.3.1.3 Interface Method Implementations

This section examines the implementation by the approximating pointer domain of interface methods relating to several important areas of functionality. In each case, code examples illustrating the operation of the pointer domain will be shown. Complete listings can be found at the location specified in Appendix 3.

12.3.1.3.1 Memory Access

Figure 161 shows the implementation of two memory access methods from the pointer approximating value interface. As discussed in Chapter 5, the approximating pointer domain is responsible for performing reads and writes across *all* value domains. As a result, while the methods shown in Figure 161 are defined at the *domain* level, they read and write objects at the *value* level. The implementation of these routines (and their pointer access analogues) relies on the type-generic routines shown in Figure 162 and Figure 163.

Figure 162 holds the code associated with cross-type reads through pointers. The *PerformRead* routine allows the user to specify the maximum number of referents with which a pointer should be associated in order for the system to explicitly model the read. This strategy is adopted in observance of the fact that a read through a pointer with many referents may exhibit significant performance overhead and in deference to TACHYON's philosophy of providing the user with fine-grained control over the cost/precision tradeoff. If the number of referents of a pointer being dereferenced is greater than this number, the top value of each domain is returned. For cases where the number of possible referents is not excessive, a read of a pointer can be implemented merely as the least upper bound of the value read from each *AtomicPtrReferent*.

```
public _IAbstractIntValue ReadInt()
{
    return((_IAbstractIntValue) this.PerformRead(s_IntAccessMechanisms));
}
public void WriteInt(_IAbstractIntValue v)
{
    this.PerformWrite((_IAbstractValue) v, s_IntAccessMechanisms);
}
```

Figure 161: Implementation of two Interface Methods Related to Memory Access

```
protected _IAbstractValue PerformRead(AccessMechanisms a)
{

if (this.CtPossibleReferents() > GlobalSettings.Instance().CtMaxPtrsToAccessWhenReading())
        return(a.TopElement());
    else
    {
        _IAbstractValue v = a.BottomElement();

        for (int i = this.CtPossibleReferents() - 1; i >= 0; i--)
            v = v.LUB(a.ReadValue(this.m_rgPossibleReferents[i]));

        return(v);
    }
}
```

Figure 162: Core Implementation for Memory Reads in the Approximating Pointer Domain

The write of a pointer is only slightly more complicated, although it lacks the capacity to streamline running time at the expense of analysis. For cases in which the pointer has only a single, precisely known, concrete referent[74], a "strong write" can be performed: The old value is overwritten with the new value. Note that the value must be recorded to the current "difference list" if no difference list entry already exists for this location.

For cases where the pointer has more than one possible concrete referent, correctness requires the system to perform a "weak write" in which the contents of *each potential referent* location has its least upper bound taken with the value being written. Note that for a referent that inherently represents more than a single location (e.g. an unknown offset in some region), the "weak write" semantics is enforced by lower levels of the implementation hierarchy rather than by the code shown.

```
protected void PerformWrite(_IAbstractValue v, AccessMechanisms a)
{
    // ****may want to incorporate IncorporateNewDiffListEntry into a more abstract
    //  interface for current diff list entry

    // we have to perform a week write if we either
    //      are not sure which values we're writing too
    //
the info on the contents of the location to which we're writing is clumped together with inf
o on a
    //      bunch of other locations


if (this.CtPossibleReferents() > 1 || this.m_rgPossibleReferents[0].Region().FRepresentsArbi
trarilyManyConcreteRegions())
    {
        for (int i = this.CtPossibleReferents() - 1; i >= 0; i--)
            if (a.WeakWriteValue(this.m_rgPossibleReferents[i], (_IAbstractValue) v))
            {
            MemoryRegion r = this.m_rgPossibleReferents[i].Region();
```

[74] Note that operationally, this condition is enforced both at the *GuidingPtrDomain* level (recognizing when the target is more than one *AtomicPtrReferent*), at the *AtomicPtrReferent* level (to recognize when a particular atomic pointer referent corresponds to more than one potential index within a MemoryRegion) and at the MemoryRegion level (recognizing when a particular memory region represents more than a single concrete memory region).

```
                if (!r.FOnCurrentDiffList())
                   {

    DiffList.CurrentDiffList().EndAppendEntry(this.m_rgPossibleReferents[i].CreateCorrespondingD
iffListEntry());
                    // should be reset when we retire this current diff list
                    r.SetFOnCurrentDiffList();
                                }
                }
        }
        else
            if (a.StrongWriteValue(this.m_rgPossibleReferents[0], (_IAbstractValue) v))
            {
                MemoryRegion r = this.m_rgPossibleReferents[0].Region();
                if (!r.FOnCurrentDiffList())
                {

    DiffList.CurrentDiffList().EndAppendEntry(this.m_rgPossibleReferents[0].CreateCorrespondingD
iffListEntry());
                // should be reset when we retire this current diff list
                r.SetFOnCurrentDiffList();
                }
            }

    }
```

Figure 163: Core Implementation for Memory Writes in the Approximating Pointer Domain

12.3.1.3.2 Least Upper Bound

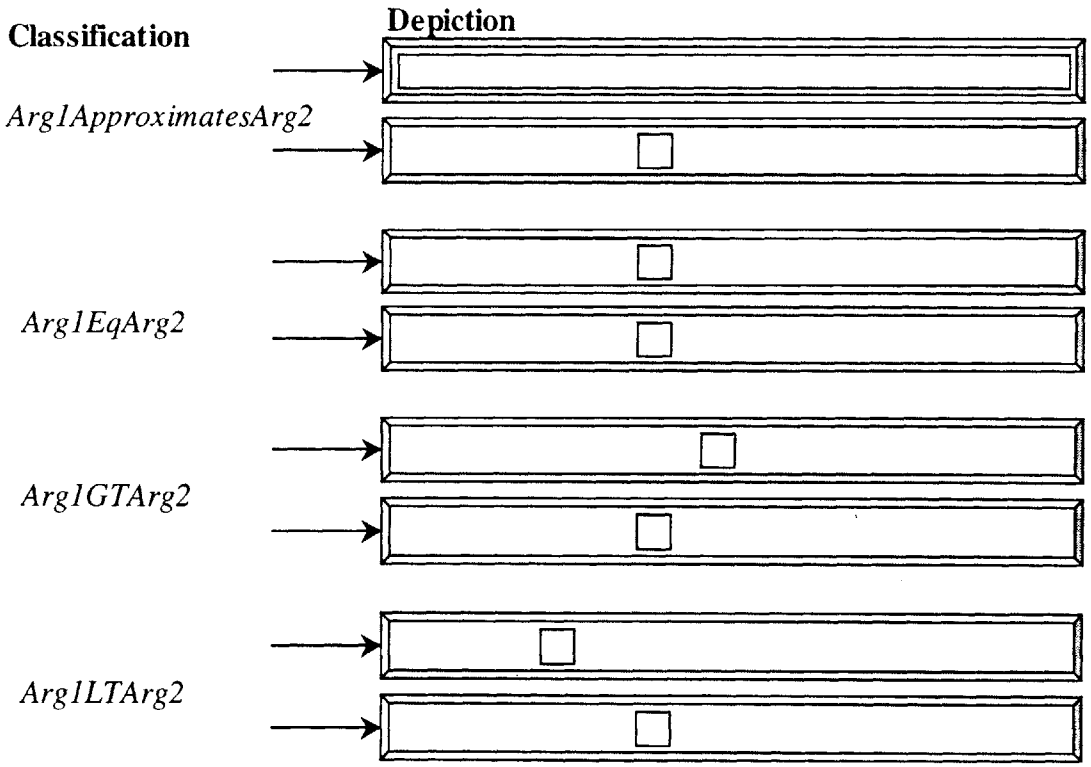

Figure 164: Possible Relationships among *AtomicPtrReferent* Objects.

The code in Figure 165 shows the implementation of the least upper bound routine of the approximating pointer value domain. The least upper bound of two elements is taken by taking the union of the sets of atomic pointer referents associated with each value. Some additional work is required to transform the results so as to be mutually disjoint and to place them in a canonical order. The central loop of the routine serves to "merge" the pointer sets held in each operand, according to the domain and offset they represent. A critical component of this is process is the call to the *AtomicPtrReferent* method *compare*, which compares different referents within the same abstract memory region. Figure 164 depicts the possible relationships among referents in a graphical form. As might be expected, the *compare* method will also play an important role in the implementation of pointer comparison interface methods.

```
  public _IPtrDomainInfo FiniteHeightLUB(_IPtrDomainInfo arg2, boolean[] fChangedOut)
  {
      // ok, this is tricky --
 we need to union together these sets, which may or not be disjoint.
      //
    in addition, we need to take into account the fact that some possible values can legally app
roximate many others
      //    need to keep in mind that these are logical "OR" of the possible values --
 if there is more
      //    than one referent, we cannot be sure to which target the write will occur.
      //
    (e.g. unknown offset for region r can be used instead of many  known offsets for region r)


      // ok, at a given point, we could EITHER have
      //
    some element from region r in one of the args and an element from a region r2 > r in the oth
er element
      //        in this case, there is no overlap and we just
      //        take the element from region r and add it to the LUBed value
      //        advance it
      //
      //    two disjoint elements from region r in each of the args
      //        in this case, we just
      //        add both to the LUBed value
      //        advance both arguments
      //
      //    two elements that overlap in each of the args
      //        if the elements are equivalent, we just
      //        take the element and add it to the LUBed value
      //        then one element must subsume the other
      //        take the element and add it to the LUBed value
      //        advance both arguments

      GuidingPtrDomain narrowedOther  = (GuidingPtrDomain) arg2;
      int advanceDirective;

      int len1 = this.CtPossibleReferents();
      int len2 = narrowedOther.CtPossibleReferents();

      // special cases we need to handle, since they won't work with the loop machinery (which a
ssumes at
      //    least one reference in each of the values

      if (len1 == 0)
      {
          fChangedOut[0] = (len2 > 0);         // LUB(this,current) > this if !(this == current)
          return(s_MRGuidingValuePtrReturnValue  = arg2);
      }
      else if (len2 == 0)
```

```
    {
        fChangedOut[0] = false;             // here, LUB(this,current)=this
        return(s_MRGuidingValuePtrReturnValue = this);
    }
    else
    {
        boolean fChanged = false;

        int i1 = 0;
        int i2 = 0;

        AtomicPtrReferent e1 = this.m_rgPossibleReferents[i1];
        AtomicPtrReferent e2 = narrowedOther.m_rgPossibleReferents[i2];

        int id1 = e1.Region().Id();
        int id2 = e2.Region().Id();

        while (i1 < len1 && i2 < len2)
        {
            if (id1 < id2)
            {

    // ok, here is a referent region in this but not in curr => we haven't added anything to t
he product
                //that is not in this => no change hhere.

                this.AddToTmpArray(e1);
                advanceDirective = AdvanceArg1;

            }
            else if (id2 < id1)
            {
                // here we have a referent region in curr but not in this  => counts as change

                fChanged = true;
                this.AddToTmpArray(e2);
                advanceDirective = AdvanceArg2;
            }
            else
            {
                // ok, here we have a referent region that is in both this and curr
                // but WITHIN the region, we may refer to different regions

                // remember that arg1 is "this"  and arg2 is "curr"
                switch (e1.compare(e2))
                {
                case AtomicPtrReferent.Arg1ApproximatesArg2:
                // no change -- we can incorporate the arg2 value into the arg1 value
                // nothing recorded yet -- just advance arg2
                advanceDirective = AdvanceArg2;
                break;
                case AtomicPtrReferent.Arg2ApproximatesArg1:
                // here we have something in curr but not in this => counts as change
                fChanged = true;
                // nothing recorded yet -- just advance arg1
                advanceDirective = AdvanceArg1;
                break;
                case AtomicPtrReferent.Arg1EqArg2:
                // here we have exactly the same pointer referent.
                // this is not viewed as a  change, since the element is in "this
                // add this to the array
                this.AddToTmpArray(e2);
                // here, advance BOTH
                advanceDirective = AdvanceArg1 | AdvanceArg2;
                break;
                case AtomicPtrReferent.Arg1GTArg2:
                // here we have something in curr but not in this  => counts as change
                fChanged = true;
                this.AddToTmpArray(e2);
                // here, e2 is smaller => must be added first
                advanceDirective = AdvanceArg2;
```

```
                    break;
                    case AtomicPtrReferent.Arg1LTArg2:
                    // ok this means something in this but not in cur  -- no change
                    this.AddToTmpArray(e1);
                    advanceDirective = AdvanceArg1;
                    break;
                    default:
                    throw new Error("unrecognized classification by AtomicPtrReferent.compare");
                    }
                }

                if ((advanceDirective & AdvanceArg1) != 0)
                {
                    i1++;
                    if (i1 < len1)
                    {
                    e1 = this.m_rgPossibleReferents[i1];
                    id1 = e1.Region().Id();
                    }
                }

                if ((advanceDirective & AdvanceArg2) != 0)
                {
                    i2++;
                    if (i2 < len2)
                    {
                    e2 = narrowedOther.m_rgPossibleReferents[i2];
                    id2 = e2.Region().Id();
                    }
                }
            }


            while (i1 < len1)    // here, all leftovers are in "this" => no change here
                this.AddToTmpArray(this.m_rgPossibleReferents[i1++]);

            while (i2 < len2)
            {
                // here we have something in curr but not in this  => counts as change
                fChanged = true;
                this.AddToTmpArray(narrowedOther.m_rgPossibleReferents[i2++]);
            }

            fChangedOut[0] = fChanged;


        GuidingPtrDomain result = new GuidingPtrDomain(s_rgReferentsTmp, s_iReferentsTmpNext, null
);
            this.ClearTmpArray();
            return s_MRGuidingValuePtrReturnValue = result;
        }
    }
```

Figure 165: Implementation of the Least Upper Bound Operator for the Pointer Approximating Value Domain

12.3.1.3.3 Offset

The handling of pointer offsets is shown in Figure 166. The implementation of the *operatorOffset* routine makes a fundamental distinction between three types of offsets: Offsets by known constant values, offsets by partially known amounts (i.e. amounts that are neither precisely unknown or completely unknown) and offsets by unknown quantities.

```
    public _IPtrDomainInfo operatorOffset(_IIntDomainInfo vOffset)
    {
```

```
        _IGuidingIntDomainInfo vNarrowed = (_IGuidingIntDomainInfo) vOffset;

        if (((_IGuidingValueDomainInfo) vNarrowed).FConcrete())
            return this.operatorAddConstantOffset(vNarrowed.SSValue());
        else
        {
          // *****note -- we could generalize this code so as to allow for more
          // precise simulation of the cases in which the offset can be tightly bounded
          // ok, if we're dealing with a non-constant value, treat it as if
          //    we don't know what offset we'll be at in each of the regions

            _IValueDomainInfo vWidened = (_IValueDomainInfo) vOffset;

            int iMaxPossibleOffset = vNarrowed.MaxPossibleSSValue();
            int iMinPossibleOffset = vNarrowed.MinPossibleSSValue();


    if (!vWidened.FIsTopElt() && GlobalSettings.Instance().FCombinatoriallyCombineGuidingPtrDom
ainAndBUOffset(this.m_rgPossibleReferents,iMinPossibleOffset, iMaxPossibleOffset))
            {
                // ok, combinatorially figure out the set of possible resulting references

                throw new Error("not yet implemented.");
            }
            else
            {
               return (s_MRGuidingValuePtrReturnValue = this.operatorAddUnknownOffset());
            }
        }
    }
```

Figure 166: Handling a Pointer Offset in the Approximating Pointer Domain.

Offsets by known constant amounts are handled by the *operatorAddConstantOffset* method, shown in Figure 168. The routine simply loops through all possible referents, and offsets each of them by the appropriate amount. Note that currently no account is taken of the possibility that the offset will result in an *illegal* offset for some of the possible referents. Such an eventuality would allow for signaling a possible error, and then allowing analysis to proceed with a pointer value that is associated with a reduced referent set. Figure 167 depicts a particular case in which this approach would be of benefit.

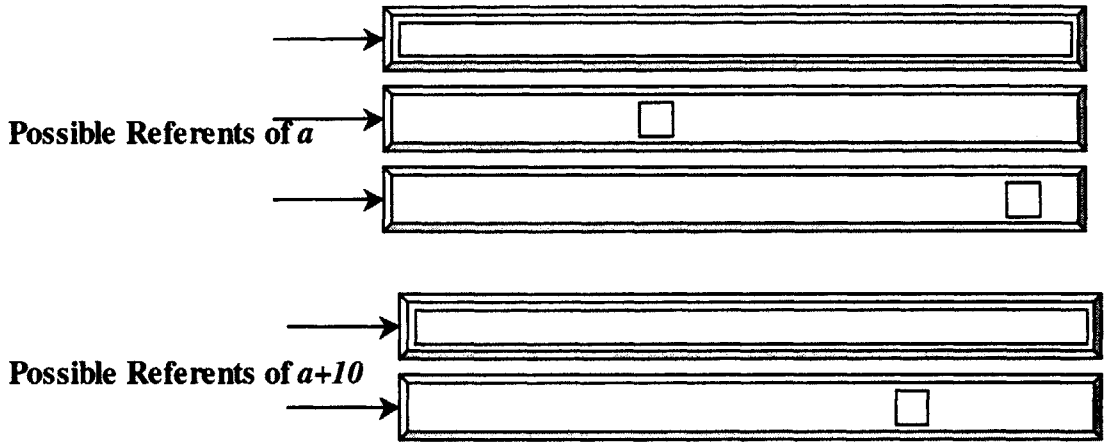

Figure 167: Instance in which Detecting Illegal Offsets can Sharpen Knowledge of Possible Pointer Referents

```
public _IPtrDomainInfo operatorAddConstantOffset(int iOffset)
{
    // note that we don't modify the existing entries (since they could be shared)
    //  instead, we create new entries
    int len =  this.CtPossibleReferents();
    AtomicPtrReferent  []rgNew;
    rgNew = (AtomicPtrReferent  []) m_rgPossibleReferents.clone();

    for (int i = 0; i < len; i++)
        rgNew[i] = this.m_rgPossibleReferents[i].operatorOffset(iOffset);

    s_MRGuidingValuePtrReturnValue = new GuidingPtrDomain(rgNew, null);
    return s_MRGuidingValuePtrReturnValue;
}
```

Figure 168: Handling of Concrete Offsets by the Approximating Pointer Domain.

Handling of unknown offsets is slightly more complicated. Because performing an unknown offset on a referent holding *any* index into a *MemoryRegion* will yield the same resulting referent (to wit, a reference to an unknown offset into that *MemoryRegion*), the system essentially must create one resulting referent for each *MemoryRegion* associated with a possible referent. This is accomplished by marking each of the *MemoryRegion*s encountered within the set of possible referents so as to guarantee that only one referent is created for a given memory region. For each yet-unmarked *MemoryRegion*, *CloneWithUnknownOffset* is to create the new referent.

```
_IPtrDomainInfo operatorAddUnknownOffset()
{

    // ok, here we are going to treat the offsets as unknown => consolidate all
    //  references to the same region .

    int len =  this.CtPossibleReferents();

    if (len > s_rgTmp.length)
        // this an upper bound on the # of elts required
        s_rgTmp = new AtomicPtrReferent[len];

    // ok, first count the number of distinct elements required
    int ctDistinctDomains = 0;
    for (int i = 0; i < len; i++)
    {
        AtomicPtrReferent r = this.m_rgPossibleReferents[i];
        if (!r.Region().FMarkedAsEncounteredInPtrConsolidation(s_idConsolidationTraversal))
        {

// ok, this is a region that we have not yet encountered.  add it to our output list

            s_rgTmp[ctDistinctDomains++] = r.CloneWithUnknownOffset();
            r.Region().MarkAsEncounteredInPtrConsolidation(s_idConsolidationTraversal);
        }
    }

    s_idConsolidationTraversal++;
    AtomicPtrReferent s_rgNew[] = new AtomicPtrReferent[ctDistinctDomains];

    for (int i = 0; i < ctDistinctDomains; i++)
        s_rgNew[i] = s_rgTmp[i];           //  copy the elements

    return (s_MRGuidingValuePtrReturnValue = new GuidingPtrDomain(s_rgNew, null));
}
```

Figure 169: Handling of Unknown Offsets in the Approximating Pointer Domain.

12.3.1.3.4 Pointer Comparison

Comparison between pointers requires a set comparison between the possible referents of one pointer and those of another. Because the referents are held in canonical order, this is less difficult than might be expected. The code for implementing one of the pointer comparison routines (for the pointer equality operator) is shown in Figure 170. In order to compare each possible referent of the pointers, this routine makes use of the same core method *compare* as did the least upper bound operator. The *compare* method indicates which of the possible relationships shown in Figure 164 exists between the referents. Each of the possible relationships is associated with a certain set of possible Boolean return values.

```
public _IIntDomainInfo operatorEQ(_IPtrDomainInfo argRHS)
  {
      GuidingPtrDomain narrowedOther = (GuidingPtrDomain) argRHS;
      int ternaryPointsToSameRegion = this.TernaryDefinitivelyPointsToSameConcreteRegion(narrowe
dOther);

      if (ternaryPointsToSameRegion == 1)
      {
          switch (this.m_rgPossibleReferents[0].compare(narrowedOther.m_rgPossibleReferents[0]))
          {
          case AtomicPtrReferent.Arg1EqArg2:
              return(s_MRGuidingValueIntReturnValue = new ConcreteQuadDomain(1));
          case AtomicPtrReferent.Arg1ApproximatesArg2:
          case AtomicPtrReferent.Arg2ApproximatesArg1:
              return(s_MRGuidingValueIntReturnValue = new TopQuadDomain());
          default:
              return(s_MRGuidingValueIntReturnValue = new ConcreteQuadDomain(0));
          }
      }
      else if (ternaryPointsToSameRegion == 0)
          return(s_MRGuidingValueIntReturnValue = new ConcreteQuadDomain(0));
      else
          return(s_MRGuidingValueIntReturnValue = new TopQuadDomain());
  }
```

Figure 170: Implementation of Pointer Comparison in the Sample Approximating Pointer Domain.

12.3.1.3.5 Pointer Creation

Finally, Figure 171 shows the approximating pointer domain's implementation of routines associated with creating pointer values. These routines are called by the analysis machinery in cases where new abstract pointer values arise in the program: When taking the address of a variable (e.g. using the "&" operator), when creating a value from the result of an initialization request, and when creating an uninitialized pointer value. In the case of allocation, the analysis machinery invokes the abstract state domain to take care of the actual allocation; a reference to the resulting region is then passed to the *CreateFromAllocation* method. Similarly, the analysis machinery provides the method handling address calculations with identification of the *MemoryRegion* and offset at which the appropriate value resides.

There is an important difference between the handling of allocation-supplied addresses in the case of allocation and when taking the address of variables. In particular, variables can be located *inside* a region (i.e. with $iOffset > 0$), while all allocations yield referents to the beginning of a region (the underlying reason behind the "0" parameter in the *AtomicPtrReferent* constructor invocation.)

```
public _IPtrDomainInfo CreateFromAddressOfVariable(_IMemoryRegion region, int iOffset, int srcLo
c)
  {
      return(s_MRGuidingValuePtrReturnValue = new GuidingPtrDomain(new AtomicPtrReferent((Memory
Region) region, iOffset)));
  }
public _IPtrDomainInfo CreateFromAllocation(_IMemoryRegion region, int srcLoc)
  {
      return(s_MRGuidingValuePtrReturnValue = new GuidingPtrDomain(new AtomicPtrReferent((Memory
Region) region, 0)));
  }

  public _IPtrDomainInfo CreateUninitialized(int srcLoc)
  {
      return(s_MRGuidingValuePtrReturnValue = new GuidingPtrDomain());
  }
```

Figure 171: Methods to Create Approximating Pointer Domain Elements.

12.3.1.4 Opportunities for Higher-Precision Analysis Based On Conditionals and Errors

12.3.1.4.1 Introduction

The pointer representation establishes that the referent of the concrete pointer or pointers for which it serves as an approximation is a member of some set of possible referents. As was briefly remarked in Section 12.3.1.3.3, certain situations encountered during analysis may eliminate some of these values as possible pointer referents, thereby making the representation more constrained and therefore more precise. As was the case for the sample space approximating domain for integers (see Section 11.4.5), although the current system does not recognize such opportunities for strengthening analysis precision, much of the machinery is in place to allow exploitation of such conditions.

12.3.1.4.2 Examples

For example, Figure 172 and Figure 173 show two cases in which such "sharpening" can occur. In Figure 172, a predicate is used to split the state into two branches based on a condition involving a pointer value. For some of the possible referents of the pointer value the predicate may be true, while for others it is false. In practical terms, this means that in one of the branches, only some of the possible referents are possible (the subset that would make the condition true), while in the other branch the complementary set of referents are possible (the subset that would make the condition false). Using the information computed in the predicate, it is possible to implement the "split" so as to "mask out" the inconsistent referents in each side of the conditional.

```
struct foo *p;
// p contains one of { null, Ref1, Ref2, Ref3 }
// q contains one of { null, Ref2, Ref4 }
if (p == q)
    {
    // here, we know that
    //   p must be ∈ {null, Ref1, Ref2 }
    //   q must be ∈ {null, Ref2 }
    …
    }
else
    {
    // here, p and q are unconstrainted
    }
```

Figure 172: Using Predicate Truth status to Sharpen Knowledge of a Value.

Similarly, we can conceptualize an *implicit* split as occurring in the context of a value operation that introduces a possible error. (See Figure 173.) For example, consider a pointer memory access operation in which one of the referents of the pointer is *null*. Such an operation can be modeled as a control flow split based on the referent of the pointer, with an exception or error condition being invoked if the pointer is null, and program flow continuing normally if it isn't. As this modeling makes clear, execution will continue only under the condition that the pointer referents do not include the value *null* – allowing the system to model the effect of the operation as "sharpening" the possible referents of the pointer.

```
struct foo *p;
// p contains one of { null, Ref1, Ref2, …, Refn-1 }
p→id = id
// here, we know that p is non-null, and must be
// one of { null, Ref1, Ref2, …, Refn-1 }
```

Figure 173: Using Error Conditions to Sharpen Knowledge of a Value.

Note that many – and perhaps most – predicates are too complicated to allow for a straightforward specification under which conditions a predicate is known to be true or false: The truth or falsehood of the predicate may depend on complicated combinations of many variables or the result of a function call. It is in general unlikely that these conditions can be compactly expressed in such a manner as to allow for deducing which pointer values are possible when the predicate is true, and which when it is false. The epistemic "sharpening" discussed here is therefore only feasible with certain restricted types of predicates.

12.3.1.4.3 Necessary Mechanisms

In order to take advantage of the opportunities for sharpening noted above, the system must use a number of mechanisms, some of which are currently implemented and some of which would needed to be added to the current system.

The *DynamicSplit* routine implemented by existing domains plays an important role in advantage of these opportunities. As was noted in Section 11.4.5, this routine allows the system to establish the state on each conditional branch as the greater lower bound of the appropriate value of the conditional and the current state entering the conditional.

What is required in addition to this routine is the ability to compute the appropriate condition within the predicate so as to allow the greater lower bound to be taken with the state on either side of the conditional. In other words, in order to recognize which pointer referents are valid on either side of the conditional, the domain implementations need to compute for which potential referents the predicate is known to be true and for which referents is known to be false. (For the case of potential error conditions, we need to recognize for which referents the error will be known to occur or not occur). The most plausible way of computing this information would be to extend the methods of the type-specific value interfaces to indicate contexts in which a *predicate* value is being computed. In such cases, the domains could internally note for which possible values of the value the predicate is definitively known to be true or false. When the greater lower bound operator is subsequently applied, the information stored away at the time that the predicate was computed could be used in performing the greater lower bound.

12.3.1.4.4 Conclusion

As was the case for the sample space approximating domain discussed in Chapter 11, the addition of support for greater analysis precision by means of more precise modeling of explicit and implicit conditionals would require only modest effort and seems an attractive addition to the current system.

12.3.1.5 Efficiency Concerns

One issue not discussed above concerns the implications of pointer domain lattice structure for analysis efficiency. Each of the sample approximating scalar domains presented in Chapter 10 and Chapter 11 adhere to a fixed-height domain structure, allowing for their use in a quadratic analysis. By contrast, the sample approximating value domain for pointers associates each pointer with a powerset abstract domain, where the elements of the set are abstract locations. As discussed in Section 12.2.6.4, the abstract state domain forms a bijection between *-Regions and points of allocation within the program text. If n is the size of the program in lines of code, this implies an $\omega(n)$ limitation on the number abstract locations present (noted above). By virtue of the powerset abstract domain used by the approximating value domain for pointers, the $\omega(n)$ limit on the number of abstract locations[75] induces an $\omega(n)$ limitation on the height of the

[75] This $O(n)$ limited on the cardinality of the set of abstract locations only applies, of course, if the user sets analysis parameters so as to perform fixed-pointing analysis of loops.

pointer domain. While the detailed modeling of pointers is unquestionably desirable, the adoption of this domain has serious worst-case implications for analysis running time: With the domain in place, the number of iterations potentially required to relax a loop to a fixed point is proportional to the number of locations times the height of the value in each location = $\omega(n)$ $\omega(n)$ = $\omega(n^2)$. Because the loop can itself be of length $\omega(n)$, the worst-case asymptotic running time rises to $\omega(n^3)$. The degree to which the high cost will be reflected in practice is unclear to the author and awaits large-scale testing.

12.4 Conclusion

This chapter has provided a brief look at sample implementation of two approximating domains associated with modeling the abstract state – the sample approximating state domain, and the sample approximating pointer value domain. These two domains work together to provide a rich model of program state whose precision can be dynamically adjusted by the user to achieve the desired accuracy/analysis time tradeoff. The abstract state abstractions presented here are distinguished from past state representations in two ways. Firstly, they allow for representations of arbitrarily high quality (ranging from models of concrete memory states to very low-resolution models of abstract state). Secondly, they differ in that they are *adaptive*, dynamically recognizing and taking advantage of opportunities to save space and analysis time in situations where the advantages of a high-fidelity representation are judged to be small.

Unfortunately, the sample implementation of the approximating pointer value domain is associated with a deep $\omega(n)$ lattice due to powerset semantics. While this provides the sample approximating pointer value domain with high accuracy, it extracts a heavy performance tool, boosting analysis time to a theoretical $\omega(n^3)$ worst-case performance. While the actual performance of past systems [Ayers 1993] suggests that empirical performance of abstract execution-based analysis algorithms is frequently far better than the worst-case performance, it remains to be seen whether the potentially poor performance of the pointer domain will represent a problem on real-life code.

Chapter 13 Conclusion

13.1 Summary of Approach

This thesis has formulated a novel approach to program analysis that is based on generic abstract interpretation. By virtue of its use of a simple, uniform model of program analysis, it is more general than traditional approaches and greatly lowers the level of expertise needed to create or customize a specific type of analysis. Informally, we might characterize it as an "object oriented" approach to analysis, in that it both componentizes the mechanics of the analysis process and places the control over the analysis with the program being analyzed.

Traditional "procedural" analysis techniques treat the program under analysis as data to an analysis function that specialized to perform a *particular* type of analysis on *any* legal program. This approach derives performance advantages from the "hard-coding" of the analysis algorithm within each such function, but only at the cost of flexibility: Performing a new variety of analysis – even a slight variation on an existing form – requires the creation of an entirely new analysis function. Moreover, because of the intricacies of formulating a sound and efficient analysis algorithm to collect a particular type of information, the design of such algorithms is an activity confined strictly to the domain of experts.

By contrast, the methodology advanced here views program analysis as the running of an abstract execution "method" of the program itself (or, more precisely, of an "abstract semantics" object derived from the program.) While traditional analyses are specialized to collect a particular, fixed type of information, the analysis method is parameterized to perform any of a broad range of analyses. But because the analysis method is always performed on the program object, it can safely gain the performance benefits of specialization to that particular program. Such specialization can be performed automatically, without need for expert involvement in the design of the analysis. The specialization process yields a "compiled analysis" algorithm that can be used to perform any of a wide variety of particular analyses. Moreover, a partitioned design fosters reuse and simple mixing of analysis semantic components, and allows a given type of analysis information to be collected in the context of a wide variety of knowledge concerning program input and the values circulating in the program.

The TACHYON system compiles a user's modules into a program that conducts a generic abstract execution algorithm on the user's program. To conduct a custom analysis, the user creates abstract value and/or state domains. By implementing a small set of compact interfaces, these domains encode the

semantic rules associated with their operation. When the user runs the program, these domains collect or approximated information as desired by the user.

The coupling of the generality of the abstract interpretation strategy with the extensibility and scalability mechanisms described in this thesis make it a simple matter to define the logic of new analysis algorithms or to customize existing algorithms.

An important goal of generic analysis is providing the user with the capacity with fine-grained control over the balance between the precision and the time requirements of an analysis. The segregation of the semantic domains that determine the information to be collected and those that determine the means of approximating run-time states and values provides the user with a straightforward means of changing this balance: By replacing one approximating domain with another. In order to allow for greater flexibility and for the adjustment of precision *during* analysis, however, the thesis further investigated a means by which the same approximating domain could support a variety of balances between analysis precision and running-time. In particular, the Disjoint Interval and Sample Space approximating value domains introduced in Chapter 11 admit to wide scaling of analysis precision. Unlike existing abstractions for capturing information concerning partially known values, the sample space representation is capable of precise modeling of the results of value *combinations* – an advantage that arises from its ability to capture information concerning inter-value dependencies. The sample space representation comes with heavy computational demands in both space and per-operation cost, but in those cases in which understanding rather than exhaustive analysis required, it offers fine-grained scalability by virtue its ability to "downsample" and thereby represent a value in a lighter-weight manner. Unfortunately, the probabilistic analysis that downsampling entails is only feasible in certain contexts.

13.2 Contributions

The fundamental contributions of this thesis have been the following:

- **Conceptualization of a Generic Abstract Execution Framework.** Abstract execution has long been acknowledged as a common conceptual framework for understanding program analyses. But the research work discussed thesis would appear to be the first aimed at making use of the shared formal underpinnings of program analysis to build a common *implementation* framework for program analyses. This thesis takes this new approach a step further by providing a framework that can be *dynamically* customized to perform different analyses through polymorphic parameterization of a generic analysis engine.

- **Formulation of a Partitioned Domain Framework.** In Chapter 3, this thesis formalized the notion of an extensible analysis framework that makes use of two types of componentized "semantic domains." The definition of such a framework and the resulting interfaces has two important benefits.

  - **Strategy allowing for independent variation of the precision of an analysis and the information to be collected.** By segregating information into the "approximating" and "collected" domains, the strategy ensures that an analysis collecting a particular sort of information can operate in the context of many different levels of precision concerning program inputs and values. In contrast to the confining static/run-time analysis dichotomy discussed in the Introduction, this approach provides the user with the ability to collect a given type of information to a wide spectrum of analysis precision/running-time tradeoffs.

  - **Mixing and Subclassing of Analyses.** A further advantage of the partitioned domain framework is the capacity to transparently plug a collected domain into the framework in order to collect the sort of information of interest to the user. This allows for a modular analysis mechanism in which semantic domains collecting different types of analysis information can be mixed and matched. This gives many of the advantages of "subclassing" an analysis without the need to make use of the source code of the analysis being subclassed.

- **Definition of the Value and State Interfaces.** The contents of Chapter 5 and Chapter 6 represent perhaps the core contribution of this thesis: The formulation of implementation interfaces that implement a basis set of primitives needed to conduct a generic partitioned analysis. Significant amounts of reformulation was needed to arrive a set of such interfaces that is both simple to implement and allows for straightforward creation of the "compiled analysis" code.

- **Compiled Analysis.** This thesis appears to be the first to examine the idea of specializing the analysis framework to the program, rather than to the information to be collected. At a time when compilers are increasingly going to heroic lengths to analyze code, the compiled analysis methodology offers a new route to improved analysis performance. The benefits of this approach accrue regardless as to whether the analysis under consideration is based on abstract execution or another methodology. Analysis systems that are currently carefully specialized with respect to the information to be collected can gain additional performance through (automatic) specialization with respect to the program to be analyzed.

- **Construction of a Working Example System.** While it is always desirable to have a working system demonstrating the ideas behind a thesis, in this case the value of building such a system goes well

beyond simple demonstration. In particular, many of the subtle issues arising in the design of the interfaces did not emerge until the software system was under construction. The implementation of the sample domains discussed in the Introduction, Chapter 9, Chapter 10, and Chapter 11 was also extremely helpful in uncovering important issues in the design of the interfaces. A substantial effort went into building the fundamental run-time implementation and sample approximating domains that were discussed in Chapter 5 and Chapter 6, much of it spent in re-implementing components once they were thought through more completely.

- **Exploration of Extensible Representations.** The Disjoint Interval and Sample Space domains discussed in Chapter 11 arose from disappointing experiences with bounded uncertainty domains in the PARTICLE [Osgood 1993] system and in the literature. While it is unclear to the author whether these representations will see use, they offer some valuable features and occupy novel and interesting points in the spectrum of value representations. In addition, they allow a user to select a desired analysis precision without having to make use of a specialized approximating value domain. This is particularly true of the sample space domain, which is especially attractive on account of its potential use as a research tool and in capturing value dependencies and frequency information in truly heavy-duty analysis.

13.3 Practicality

While this thesis has not had the opportunity to completely finish the implementation of the envisioned system, enough pieces have been built to serve as convincing evidence that the fundamental strategy is not only sound but likely practical as well.

The methodology used to generate the extensible analysis framework is straightforward and requires only simple compiler transformations. With the naïve translation in place, the resulting code can undergo substantial "bloat", but with a bit of refinement would be well inside the realm of reasonable use.

As is the case for any analysis algorithm, the abstract value semantics must be chosen with some care in order to allow for analysis that is not only sound, but timely as well. The TACHYON methodology is no exception here, but does allow for much simpler and safer reuse of existing, tested analysis strategies. Once a particular abstract domain and associated semantics has been implemented and debugged, it is available for easy, "snap-in" use within any future analysis. But this process cannot be done willy-nilly: The user must take care to avoid constructing an abstract value domain that is too high, as it can lead to long convergence times. In addition, the use of a representation such as the sample space abstraction can lead to undesirably long space and per-operation time costs. An important lacuna in this thesis is the absence of

medium and large-scale empirical tests of analysis performance using this methodology. Chapter 7 gave some preliminary theoretical worst-case bounds on analysis running-time, but such theoretical characterizations of performance often bear only glancing resemblance to average-case execution profiles [Ayers 1993] [Chen 1994]. While the structure of user-defined domains strongly impacts analysis convergence time, empirical measurements of base-line performance are sorely needed.

Despite the apparent feasibility of running the generic analysis algorithm, it is clear that the naïve abstract execution algorithm employed has heavy performance costs relative to the efficient worklist algorithms featured in recent contributions in the abstract execution area [Ayers 1993] [Chen 1994]. Such systems take strong advantage of the type of information to be collected to streamline the analysis algorithm. Because semantic flexibility is the defining feature of the generic analysis approach, this option is not open to us. It seems unavoidable that the generic analysis engine will suffer some performance disadvantages on account of this generality.

13.4 Avenues for Further Research

A central goal of this thesis has been increasing the flexibility of analysis – allowing the user to vary the precision and information collected by a particular analysis. Some of the most important promising (or pressing!) avenues to future work are mentioned below.

- **Performing quantitative measurements.** A clear imperative within the current study is to clarify the performance implications of the TACHYON strategy with extensive performance measurement. A few issues for which benchmarking would be of particular interest are given below:

    - **Comparison of performance of interpreted vs. TACHYON's compiled approach.** As was noted in Chapter 7, both traditional analysis frameworks and TACHYON represent specialized forms of analysis – they are simply specialized to different invariants. While the compiled analysis described in this thesis avoids the interpretive overhead associated with determining program structure, it carries with it a less obvious sort of interpretive overhead: That arising from leaving dynamic the particular domains that are present during operation. As noted in Chapter 3, this interpretive overhead is reflected not only in the dynamic binding of domain-specific method calls, but more subtly in the inability of the analysis algorithm to make use of a "worklist" algorithm such as those of [Chen 1994] and [Ayers 1993]. While the stochastic model of Section 7.3.3.3 suggests that compilation of the analysis is likely to yield performance benefits that more than compensate for the overhead entailed by the polymorphism in the

analysis algorithm, such results are preliminary in the extreme and need to be measured empirically.

- **Relative Performance of "Direct Execution" vs. Prestate-Caching Mechanisms.** One implementation option not examined herein is the performance benefit extending from the use of mechanisms that cache prestate/poststate approximations during fixed-point analysis. [Chen 1994] illustrates a model for this technique that might be transplanted into TACHYON. The implementation of such mechanisms within an approximating abstract domain could yield strong performance enhancements.

- **Replacement of Exception-Based Bottom State Signaling with Lighter-Weight Approaches.** As introduced in Section 8.4.1, the exception-based approaches offer convenience and clarity, but is likely to be associated with a substantial performance burden. Eliminating such structures in favor of a less heavyweight approach constitutes a clear route to improved performance.

- **Specialization Tradeoffs.** A key element of the TACHYON strategy is the jettisoning of the traditional approach of specializing an analysis algorithm to an abstract domain in favor of a methodology that specializes that algorithm to a particular *program*. In the absence of detailed measurements on larger programs, the net performance result of this tradeoff remains to be seen.

- **Achieving Semantic Extensibility in Worklist-Based Abstract Interpretation.** Some of the most important recent advances in the area of abstract execution have been the formulation of efficient queue-based algorithms for collecting information on program behavior. [Ayers 1993; Chen 1994] These algorithms offer strong empirical performance benefits compared to the naïve abstract execution methodology used in TACHYON and previous systems, but achieve those benefits in large part by carefully specializing the algorithm to the particular type of information being collected, (a tendency particularly pronounced in [Ayers 1993]) and by constraining certain aspects of the semantics domains (e.g. the semantics of assignment in [Chen 1994]). If queue-based methods could be equally well applied in the context of arbitrary semantic domains, and with fewer constraints on the behavior of those domains, they offer the potential for significantly improving the performance of generic abstract execution.

- **Elimination of Domain Modification during Analysis.** The current TACHYON run-time system makes use of a technique designed to allow users to modify the set of collecting domains during analysis. It seems unlikely indeed whether the flexibility so obtained justifies the imposition of a

substantial cost on each operator combining together two or more values and states. Domain-specific means of enabling/disabling collection of information by domain elements would provide similar functionality at much lower run-time cost. The use of such expensive and overly general machinery should be discontinued.

- **Development of More General Approximating Domain Interfaces.** The segregation of semantic domains into "approximating" and "collected" domains offers some very strong benefits for analysis flexibility. By requiring all collecting domains to access approximating domains through abstract interfaces, the implementation of the approximating domains can be changed without the need to modify the collected domains that depend on them. Unfortunately, the current interfaces supported by approximating domains are very simple, and offer the collected domains quite impoverished information on approximating domain contents. In effect, collected domains are allowed to make use of no more than information regarding the maximum and minimum values taken on by a value. This is useful in that it provides a very useful "least common denominator" type of information concerning approximating values, but has a significant drawback in that it allows no means to access richer information when it is available. By contrast, many collected domains seek to make use of as much information as possible regarding the approximating information that is present. Knowledge of the maximum and minimum values taken on by a variable is frequently an impoverished proxy for the full information maintained by the approximating domains. As a result, the creator collecting domains is tempted to create "special case" handling of different types of approximating domains – a strategy that provides greater information to the collecting domain only at the cost of weakening the approximating/collecting domain distinction. One possible way to deal with this tension between abstraction and the amount of information available is to require approximating domains to implement a richer interface, one that allows access to a more detailed approximations when they are possible. It is unclear to the author what form this interface would take, but the observations made in Appendix 1 on the character of uncertainty during analysis may provide a starting point. An alternative, more pragmatic, route would be to aim for the development of value representations that are themselves highly scalable. This would allow for the same "special case" in the collected domain code to handle a wide variety in analysis precision.

- **Investigation of further means to partial value representations**. As demonstrated in Appendix 2, existing partially known value representations typically lose information rapidly in the presence of value combinations. The sample space approximating domain presented in Chapter 11 avoids this difficulty, but is associated with a heavy computational overhead at almost any precision. Given these

shortcomings, it seems worthwhile to consider further investigations into mechanisms that would permit capturing information on partially known values. An obvious starting point would be to further investigate the Disjoint Interval domain also presented in Chapter 11, the precision and performance implications of which has been only glancingly examined in this thesis. Another clear priority is to examine the potential of predicate-based representations in greater detail. While the discussion in Appendix 2 above highlighted some substantial challenges faced by such approaches, it only dealt with this class of representations in the most superficial way. Of particular interest in this area are powerful approaches in which the predicate language is Turing Complete (such strategies are hinted at by some previous work, such as [Nguyen 1989]).

13.5 Closing

The extensible, generic analysis approach offers many usability advantages over traditional models. Given the widespread and pronounced trends towards increasing componentization, product customizability and the growth in analysis tools, it seems inevitable that this model will play an important role in the evolution of programming environments. It is hoped that the methodology articulated in this thesis will contribute towards greater analysis flexibility and usability among all those who need to understand code.

APPENDICES

Appendix 1 Uncertainty in Program Operation and Analysis

1.1 Introduction : The Roots of Uncertainty

In this chapter, we turn to examining the character of uncertainty that arises in the context of program analysis: How it arises, the mechanisms and qualitative manner in which it is propagated, and the ways in which degrades the quality of the analysis that is performed. Despite the ubiquity of uncertainty and the familiarity of many of these concepts to compiler designers, it is hoped that a systematic taxonomy and examination of uncertainty will help expose some oft overlooked insights. Our interest here is on the manner in which *a priori* uncertainty concerning program inputs (or, more broadly, execution context) entails uncertainty concerning all aspects of program operation. Our analysis is thus aimed at providing an understanding of the upper limits on knowledge of the behavior of a program given a certain probability distribution of inputs, much as a execution time analysis can reveal inherent lower bounds on the running time of any algorithm that attempts to accomplish some task.

Most software developers carry around simple mental models of how uncertainty concerning program quantities affects other quantities during analysis. For example, it is readily understood that if the analysis system has little or no knowledge of the values associated with one program quantity, the results of combining that quantity with others will likely be imprecisely known as well. Our interest here lies much deeper: We are interested in investigating the nature of uncertainty in program analysis: How it originates, how it is affected by the structure of the program and the algorithms employed therein, and how it is manifested to the user at different levels of modeling precision.

Our method of approach to this problem is a classical one: First, we characterize the domain at the finest grain level possible. In this case, we examine in detail the ways in which program data values at a given point in the execution of the program relate to sources of uncertainty in the program. Based on this understanding of the character of program values, we turn to examine how such values combine during an operation so as to yield a resulting value. We then proceed to an understanding of how macroscopic patterns – some familiar, some ubiquitous but rarely observed – emerge from compositions of such interactions.

1.1.1 Introduction

Introductory books on programming and language semantics are filled with programs whose operation is completely specified by their code. Reality is rarely so simple. In particular, programs are typically

parameterized so as to vary their behavior according certain characteristics of the execution context. For example, programs frequently accept user input through keyboard or mouse activity, read information from persistent stores such as databases or files, carry on interactions over a network, and make use of dynamic system information such as the value of the system clock. Using a common term in the broadest sense, we will call the aspects of execution context to which a program is sensitive the "input" to that program.

The finest-grained model of all possible distinguishable execution contexts in which a program will be run can be characterized as a *sample space*, with each possible execution context associated with a probability of occurrence, and implying a particular (potentially infinite) *sequence of states* of the program.

Taken by itself, this approach to thinking of programs in the context of uncertainty concerning program *inputs* lends little insight into how that ignorance ramifies into uncertainty concerning program values: The effects of each possible execution are envisioned in disjoint execution sequences, and cannot be easily compared. To get a more complete understanding of the effects of differences in execution context, we need to compare these sequences – in effect, to *simultaneously* simulate several of these possible paths of execution.

One useful way of visualizing such simultaneous execution is by considering the *superposition* of the standard semantics executions associated with each possible sample space configuration. At any point during program execution, a particular program value has separate concrete values[76] associated with each possible sample space configuration (i.e. execution context). In other words, such a value represents a *random variable* defined over the sample space of possible execution contexts. The sequence of "superimposed" program states can be viewed as a *stochastic process*, with the control flow sequence and the contents of each successive state being determined probabilistically. This "cross-sectional" perspective of program values as random variables and the program execution as a stochastic process is one of the central principles underlying the sample space representation introduced in Chapter 11.

1.1.2 Superposition: A Closer Look

While the notion of "superimposing" distinct execution contexts is a natural one, we need to provide some critical details on how that superposition is defined. To arrive at such a superimposed view of program executions, we define a "generalized" sequence of program points that safely approximates each of the sequences of program points associated with particular execution contexts. For every program point within the generalized sequence, we create a generalized state that consists of the "superimposition" of the states of

[76] Technically, the configurations may also be associated with the distinguished bottom element $\perp$ (or a type-specific element $\perp_T$ if one is considering a program in which values can take on different types T).

all corresponding program points within the standard semantics executions. Table 35 and Table 36 illustrate this principle for a simple fragment of straight-line code.

Statement	Prestate for Statement																			
	Execution Context 1				Execution Context 2				Execution Context 3				Execution Context 4				Execution Context 5			
	a	b	t	acc	a	b	t	acc	a	b	t	acc	a	b	t	acc	a	b	t	Acc
t=a*b	1	1	0	0	1	2	0	0	2	1	0	0	2	2	0	0	1	3	0	0
acc = acc+t	1	1	1	0	1	2	2	0	2	1	2	0	2	2	4	0	1	3	3	0
B=t*b	1	1	1	1	1	2	2	2	2	1	2	2	2	2	4	4	1	3	3	3
...	1	1	1	1	1	4	2	2	2	2	2	2	2	8	4	4	1	9	3	3

Table 35: Depiction of the Sequence of States During Program Execution for Each of a series of Execution Contexts.

Statement	Prestate for Statement																			
	a					b					t					acc				
Execution Context	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
t=a*b	1	1	2	2	1	1	2	1	2	3	0	0	0	0	0	0	0	0	0	0
Acc=acc+t	1	1	2	2	1	1	2	1	2	3	1	2	2	4	3	0	0	0	0	0
b=t*b	1	1	2	2	1	1	2	1	2	3	1	2	2	4	3	1	2	2	4	3
...	1	1	2	2	1	1	4	2	8	9	1	2	2	4	3	1	2	2	4	3

Table 36: Superimposed Execution as a Cross-Sectional View of the Sequence of States During Program Execution for a Series of Execution Contexts.

More formally, suppose that we are given a sample space consisting of *n* possible execution contexts $e_1 \ldots e_n$. Each such execution context e_i is associated with an execution that can itself be described as a countable series of states S_i. Each state $s_{i,j}$ is associated with a particular instruction pointer $ip_{i,j}$; thus, S_i is associated with a sequence of instruction pointers (or "program points") P_i.

Given the sequences S_i and P_i $\forall 1 \leq i \leq n$, we can define a generalized state sequence S and (a derivative) instruction sequence P. S represents the "superimposed" states that obtain at each point of the "superimposed execution," and P characterizes the sequence of instruction pointers present in S. In particular, P consists of instruction pointers $ip_0 \ldots ip_{|P|-1}$ such that $\forall$ P_i, P_i, is a (possibly non-contiguous) subsequence of P.[77]

[77] We term P_i as a subsequence of P iff $\exists$ a mapping *f* from the $ip_{i,j}$ to the ip_k such that $f(ip_{i,j})=f(ip_{i,a}) \Rightarrow a=j$ & $f(ip_{i,j})=ip_k \Rightarrow f(ip_{i,j+1})=ip_m$, $m>k$. (i.e. *f* is an injection and *f* preserves the ordering of the mapped elements). Intuitively, this means that any sequence P_i will be represented in P, although possibly with intermediate instruction points placed between any two $p_{i,j}$.

For each mapping from $f(ip_{i,j}) \Rightarrow ip_k$, the concrete state $s_{i,j} \in$ the superimposed state s_k. In other words, if a particular point p in a standard semantics execution is associated with a specific program point q in the superimposed execution sequence, then the concrete state that applies at point p is guaranteed to be part of the superimposed state that obtains at point q in the superimposed execution sequence.

In the context of straight-line code, superimposing multiple paths of standard-semantics execution is straightforward, and lends an immediate intuitive understanding to the character of the associated abstract semantics defined over random variables. In particular, all instruction sequences P_i are the same, and f can be defined as the identity function. Because there is a bijection between the states in any particular standard semantics execution and the states in the superimposed execution, the superimposed states can be collected together in a simple-minded manner. Table 36 shows the simplicity of the mapping in this case.

In the context of programs in which there are differences in control flow between different execution contexts, the notion of "superimposing" multiple paths of execution is less intuitive. This complication arises from the fact that the different execution paths do not uniformly overlap: Certain program points ip_i may be included in the paths of execution associated with certain execution contexts but may be absent from the executions taking part in other execution contexts. (See Table 37.)

```
t=a*b
acc=acc+t
if(acc>2)
    b=t*b
```

Figure 174: A Fragment of Code Exhibiting Conditional Execution.

Statement	Prestate for Statement																			
	a					b					t					acc				
Execution Context	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
t=a*b	1	1	2	2	1	1	2	1	2	3	0	0	0	0	0	0	0	0	0	0
acc=acc+t	1	1	2	2	1	1	2	1	2	3	1	2	2	4	3	0	0	0	0	0
if(acc>2)	1	1	2	2	1	1	2	1	2	3	1	2	2	4	3	1	2	2	4	3
b=t*b (In conditional)	⊥	⊥	⊥	2	1	⊥	⊥	⊥	8	9	⊥	⊥	⊥	4	3	⊥	⊥	⊥	4	3

Table 37: Non-Execution in the Superposition Model of Execution.

While execution in a standard semantics program follows a particular completely deterministic path, a "superimposed" execution can involve multiple possible paths of execution, each associated with different sets of execution contexts to which they apply. For our purposes here, the presence of multiple possible execution paths within the superimposed concrete execution paths has one significant consequence: In order

to represent program values as random variables, a slight adjustment is necessary to the domain of values for the random variables. In particular, it is important to note that at a particular point during abstract semantics execution the random variable associated with particular program value may apply only to a particular subset of possible execution contexts. (For an example of this, see Table 37.) In the context of a superimposed path of execution that is *not* taken by a specific execution context, it makes no sense to ask for the "value" of a program quantity for that execution context – the value is not associated with a value for that execution context. We can denote the absence of a value for a particular execution context by "lifting" the value domain and permitting the random variables to associate a distinguished "no value" element (symbolized as $\perp$) with particular execution contexts. Table 37 illustrates the result of lifting the value domain within the program fragment illustrated in Figure 174.

1.1.2.1 Conclusion

This section introduced the characterization of the set of program execution contexts as a sample space, and focused on a "superimposed" view of the program execution sequences that are associated with the different execution contexts. Within this superimposed view, the program is associated with a (non-unique) "generalized execution sequence" that approximates the execution sequence for any particular execution context, and program values represent *random variables* defined over the sample space of possible execution contexts. The range of these random variables is the concrete domain lifted to represent the distinguished "no value" element.

1.1.3 Value Sample and Event Spaces

1.1.3.1 Introduction

While characterizing program values and program operation from the perspective of *sample space* affords clarity in understanding the dynamics of execution, it is not a common viewpoint and is rather distant from the experience of software developers. In many cases, it is more convenient to consider stochastic computations from an *event space* perspective.

All random variables define event spaces – partitionings of the sample space into equivalence classes, each associated with the assignment of a particular value by the random variable. In the case of the random variables associated with program values, such event spaces "clump together" program execution contexts in which the particular concrete (or lifted-concrete) value associated with the program value) takes on the same concrete (lifted-concrete) value. Figure 175 shows an example of a program value's event space and its relation to the underlying sample space.

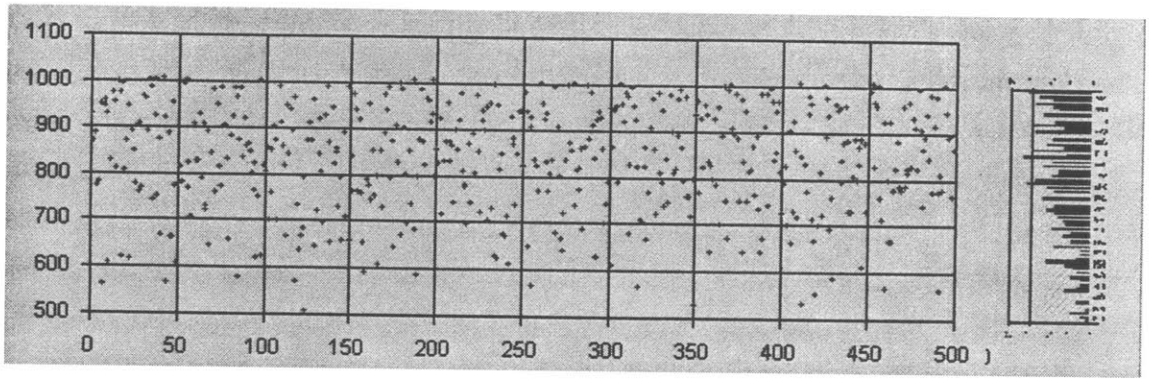

Figure 175: Illustration of the Connection between a Value's Sample and Event Space.

Figure 176 illustrates the event space of a random variable in which the horizontal axis is composed of each value taken on by a random variable and where the vertical axis corresponds to the probability of the random variable taking on each possible value. Speaking informally, if we were to draw a very large set of samples from the random variable, we would expect the histogram of the sample values to closely resemble this graphical depiction of the random variable. For the particular case of program values, such a description of the event space has a yet more familiar meaning: It approximates the histogram that would be seen if the program were to be run many times and the value of the program value examined and tallied for each such execution.

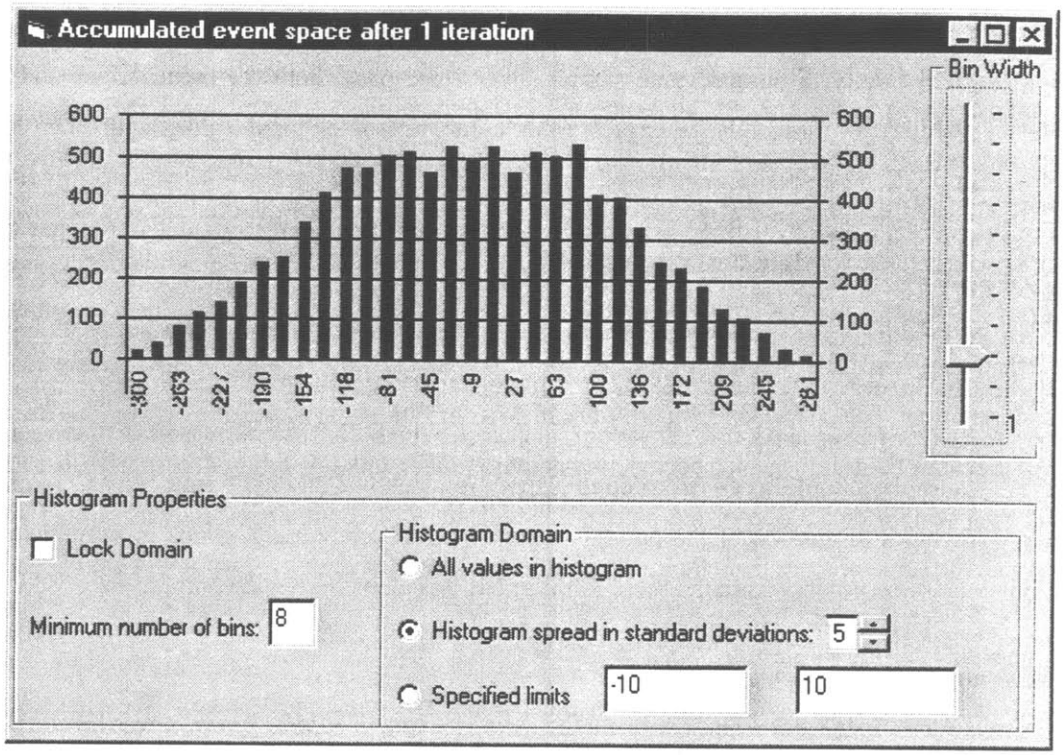

Figure 176: Sample Illustration of the Event Space of a Program Value

1.1.3.2 Summary

In summary, given a probability distribution of possible input contexts, we can completely characterize a program value by describing its assignment of values to sample space configuration. Such a description is complete and is extremely convenient for understanding the manner in which that value interacts with other values. But a more familiar mode of description can result from looking at program values from the perspective of their event space. From this viewpoint, we can get an immediate sense of the likelihood with which the program value will take on a particular lifted-concrete value, and of the histogram of lifted-concrete values that is likely to be taken on by the program value given a large number of executions of the program.

The next section uses this conceptual framework to take a further step towards our goal of understanding how uncertainty concerning the program input affects the levels of certainty that are possible about aspects of program operation. In particular, that section examines the manner in which program values combine together and the patterns induced by that combination , as viewed from the perspective of both sample space and event space.

1.2 Epistemic Dynamics

The previous section briefly discussed value sample and event space, both for input values and internal program values (which can ultimately be expressed as combinations of input values and other constants from the program text). The section mentioned the event spaces associated with internal program values that result from combinations of other program values, but gave no feel for the qualitative manner in which these event spaces reflect the values that give rise to them. This section attempts to lend a "gut feel" for the combinatorics of random variable combination, with an eye towards fostering an understanding of the broader patterns that emerge from the repeated combinations that occur in realistic code. As always, we can observe manipulations performed on random variables from either the sample space or the event space perspective. Because the sample space perspective is more fundamental and more transparent, we adopt this view first. Following a discussion of the patterns associated with combining values in sample space, we turn to examine the same process from the more familiar but less elementary viewpoint of event space.

1.2.1 Combining Values in Sample Space

This section examines value combination from the perspective of sample space. We first set out the fundamentals of such combination, which lend themselves to extremely simple characterization in sample space. The simplicity of this relationship is the primary motivating factor in adopting the sample space perspective: It allows viewing operations on complex and rich values as the superposition of many very simple operations.

We will then move beyond the result of a single operation and will examine the broader patterns associated with value combination in sample space. These patterns form the basis for understanding the effects of value combination in event space that are discussed in the next section.

1.2.1.1 Elementary Relationships

Suppose we wish to analyze program behavior given a model of the frequency distribution of likely inputs to that program. Suppose further that during analysis the expression $a \oplus b$ is encountered (where a and b represent program values and $\oplus$ stands for any standard binary operator).

Because a and b are program values, they are associated with random variables (call them A : S→Value and B : S→Value), each of which is associated with a mapping from a single unified sample space S to values. The process of combining a and b in sample space is very simple:

---

[78] Note that for reasons that will be discussed later, we must also consider the domain element $\perp$.

Let a and b be two program values defined over sample space of execution contexts S
Let $\oplus_{\perp}$ be a standard semantics operator extended to return $\perp$ for any argument that is $\perp$
Let $v(e)$ represents the value of program value v for execution context $e \in$ S.
The value $c = a \oplus b$ is a random variable defined on sample space S, and has values defined as
$c(e) = a(e) \oplus b(e)$ $\forall$ execution contexts $e \in$ S.

In other words, the result of combining $a \oplus b$ for program values a and b is the superposition of $a \oplus b$ for each possible execution context. More informally yet, we can say that *A combination of superpositions is simply a superposition of combinations.*

At an intuitive level, what this means is that viewing value combination from the perspective of sample space allows us to treat the combination of two program values defined over a sample space as the superposition of the combinations of the values of each of these program values for every possible sample space configuration.[79]

Before moving on, it is worth noting that while the sample space perspective lends a simple understanding of the relationship between the operands and results of applying a program operator, it can also reveal patterns resulting from combining two values each associated with patterns of their own. In particular, program values will frequently vary with respect to *different* aspects of program sample space. (Indeed, models of sample spaces frequently admit to multidimensional characterization, and program values may vary with respect to different of these dimensions). Suppose we are combining two program quantities that are associated with uncertain values and are essentially unrelated (in the sense that they draw from different domains of uncertainty). We would expect the result of the combination to not only itself be uncertain, but also to have the potential to take on any value resulting from combining any of the values associated with one of the quantities with any value associated with the other quantity.

While the sample space decomposition of a value provides an easy way of understanding the effects of executing program constructs on partially known values, it is not always the most desirable view through which to summarize program execution. In a similar way, the user of program analysis are frequently more interested in characteristics of a value's event space than in the sample space picture of that value. Having

[79] The sample space view of an incompletely known value can be thought of as specifying the decomposition of that value into a basis set of "eigenvectors" – particular execution contexts that remain independent through each step of analysis. The program value decomposed in this manner is associated with a particular component for each eigenvector (this is simply the value taken on by the program quantity for that execution context.) Executing any program expression on one of those eigenvalues yields back a scaled version of the value associated with that eigenvector (i.e. another value within the same execution context). The attractiveness of this basis set is based on the fact that the result of the combination simply represents the superposition of the combinations undertaken for each execution context.

given an introduction into value combination from the perspective of sample space, we turn now to understanding the effects of such combinations in event space.

1.2.2 Value Combination: The Event Space Perspective

Recall that the event space of a random variable consists of the partitioning of the sample space in which the equivalence classes of the partition consist of the sets of sample space configurations that are associated with a particular value by the random variable. Our interest now lies in characterizing the effect of value combination in event space. In particular, given the *event spaces* associated with one or more program values, we would like to have a sense of the manner in which these values are combined by an operator to produce a resulting value, and a feel for how the event space of that resulting value depends on the event spaces of the operands.

Unfortunately, in general there is insufficient information to predict the event space associated with the result of a combination from the event spaces associated with their operands. In particular, the standard event space for a program value gives no indication of the execution context associated with each of the possible values; without this contextual knowledge, any number of different event spaces could be consistent with the combination. To illustrate this point, Figure 178 and Figure 179 show the event-space resulting from two separate combinations of pairs of values – $a*a$ and $a*b$. Although the operands a and b in each combination have identical event spaces, the event spaces of the results differ markedly.

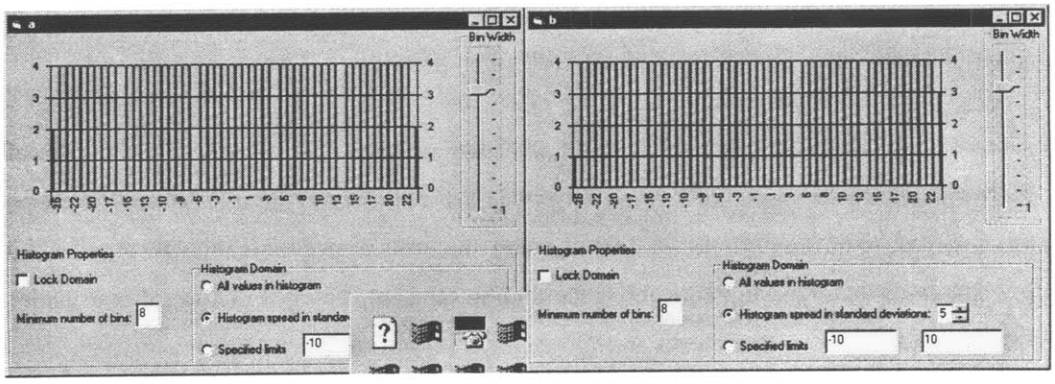

Figure 177: Two Program Values Associated with Identical Event Space Characteristics

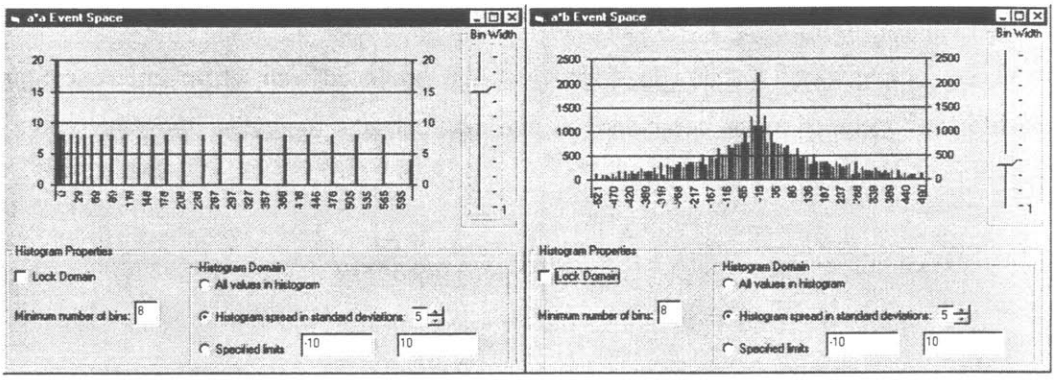

Figure 178: Results of Applying * Operator To Pairs of Operands with Identical Event Space Characterization.

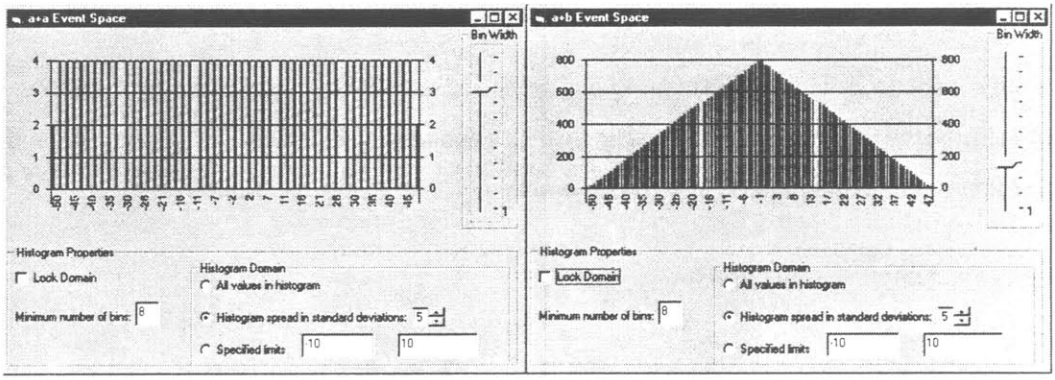

Figure 179: Results of Applying + Operator To Pairs of Operands with Identical Event Space Characterization.

In order to understand combinations in event space, it is thus necessary to borrow from the perspective of sample space. In particular, we need to recall the relation between event and sample space.

Figure 175 provides a graphical depiction of aspects of the connections between sample and event spaces for a random variable. As can be seen from the diagram, the probability of occurrence of any particular configuration v in the event space of a random variable can be calculated by summing the probability associated with all sample space configurations for which the random variable has the value v. The last section described how to calculate the sample space of a program value produced by a combination of other values. The event space of such a value can be derived directly from the sample space, and the probabilities associated with that event space can be calculated by summing the probabilities of the resulting sample space.

The random variable resulting from a combination of program values that vary with respect to different dimensions in the sample space has sample space elements associated with all possible combinations of values in each of the operands to the combination. We now consider the event space for such a random variable.

We will now approach this slightly more formally. First, we need to establish a few notational conventions. Suppose we have sample space S composed of configurations e, each of which is associated with probability of occurrence $\Pr(e)$. Given random variables $x : S \rightarrow \text{Value}$ and $y : S \rightarrow \text{Value}$ we define a single or joint probability mass function for x as follows:

$$p_x(x_0) = \sum_{e \in S | x(e)=x_0} \Pr(e)$$

The probability associated with a particular event space value v is the sum of the probabilities of each configuration in the sample space of the result that is associated with value v – that is, the sum of the probabilities of each combination of values from the operands that yields the result v.

And the joint probability mass function for (x,y) as

$$p_{x,y}(x_0, y_0) = \sum_{e \in S | x(e)=x_0 \& y(e)=y_0} \Pr(e)$$

Given $r = a \oplus b$, we define its "left inverse"[80] $r \oplus_L^{-1} a = \{ b \mid a \oplus b = r \}$

Having laid down the rules of the notation, now suppose that we have two program values a and b defined over sample space S.

If the program creates a value $r = a \oplus b$, we can define we can describe the probability mass distribution $p_r(r_0)$ for the resulting value r as

$$p_r(r_0) = \sum_{e \in S | x(e) \oplus y(e)=r_0} \Pr(e) = \sum_{v \in \text{Value}} p_{a,b}\left(v, r_0 \oplus_L^{-1} v\right)$$

[80] Note that we use the notation $\oplus_L$ in order to distinguish the *left* inverse from the *right* inverse ($\oplus_R^{-1}$, defined such that $r \oplus_R^{-1} b = \{ a \mid a \oplus b = r \}$.

Phrased in another way, when the two values being combined are independent the probability of receiving a value r_0 as a result of the operation is simply the sum of the probabilities of all of the possible pairs of values a_0 and b_0 such that $a_0 \oplus^{-1} b_0 = r_0$. Note that we are slightly overloading $\oplus_L^{-1}$ here: For the cases where $r_0 \oplus_L^{-1} v$ is unique, we treat this expression as a single value in the equation above. If $r_0 \oplus_L^{-1} v$ is *non*-unique, we implicitly extend the summation above over all possible values for f $r_0 \oplus_L^{-1} v$.

Now suppose that a and b are independent. Then $p_{a,b}(a_0,b_0) = p_a(a_0)p_b(b_0)$ and we have

$$p_r(r_0) = \sum_{v \in \text{Value}} p_{a,b}\left(v, r_0 \oplus_L^{-1} v\right) = \sum_{v \in \text{Value}} p_a(v) p_b\left(r_0 \oplus_L^{-1} v\right)$$

If the form of the expression for $p_r(r_0)$ looks familiar, it is for good reason: It closely resembles the formula for computing the convolution $p_a * p_b$. Indeed, if we choose to make f the addition operator, we arrive at the formula for the convolution.

$$p_r(r_0) = \sum_{v \in \text{Value}} p_a(v) p_b\left(r_0 \oplus_L^{-1} v\right) = \sum_{v \in \text{Value}} p_a(v) p_b(r_0 - v)$$

Other operators $\oplus$ yield formulas for $p_r(r_0)$ that are of similar flavor but which differ in their details. Figure 178 and Figure 179 show histograms representing the frequencies of occurrence of values for $p_r(r_0)$ given uniform $p_a(a_0)$ and $p_b(b_0)$ and two different operators $\oplus$.

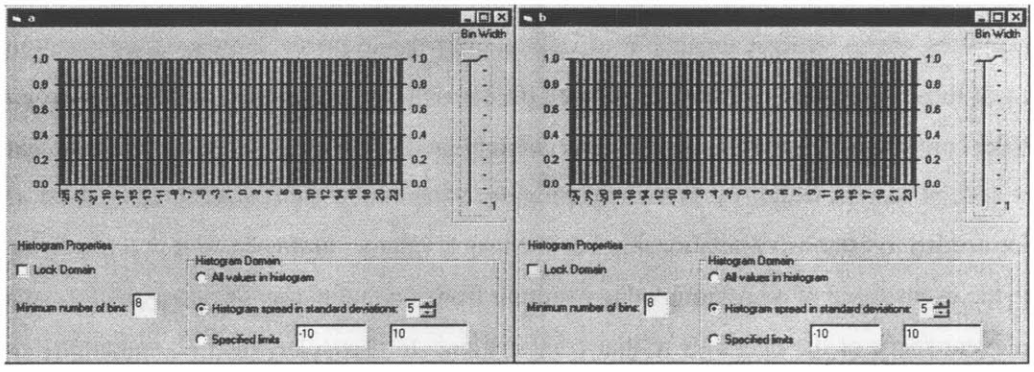

Figure 180: Uniform Event Space of (Independent) Variables a and b

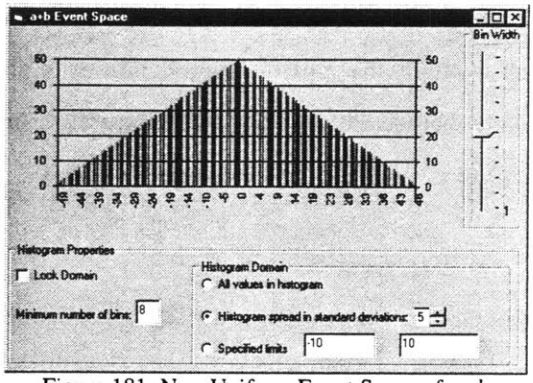

Figure 181: Non-Uniform Event Space of *a+b*

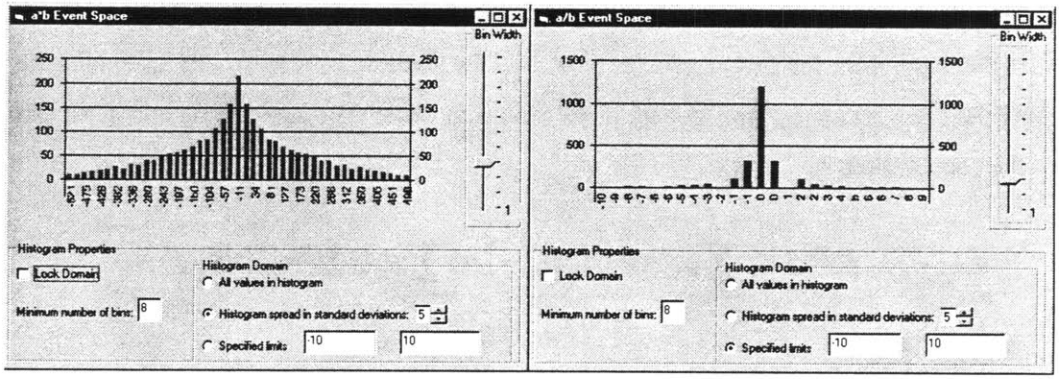

Figure 182: Non-Uniform Event Spaces of *a*b* and *a/b*

It is worth pausing to consider what has been shown in a broader light: Event space characterizations of program quantities are in general incapable of determining the event or sample space resulting from a combination of those quantities. Typically we need the far richer sample-space picture of a program value to establish the contextual information needed to accurately predict the outcome of such a combination. But for the very special case in which we know two program values to be independent, context information is therefore not needed to establish a relationship between such values. In this case, it is typically possible to characterize the event space of a program value resulting from a combination of two (or more) values based only on the event space of the operands to that combination. In particular, such combinations yield event spaces with "averaging" characteristics resulting from the statistical effects of combining independently selected samples from the event spaces of the operands. As will be shown in the next section, these statistical effects have a profound effect on the distributions of values circulating in a program.

1.2.3 Example of combinations

1.2.4 Patterns in Program Evolution

The first sections in this chapter have discussed the conceptual framework of executing a program with respect to a distribution of input, and the characterization of program values as random variables. We then continued on to discuss in some detail the means in which those values combine, and analyzed how this combination manifests itself both the sample space and event space. This section is rather different in orientation, being more empirical than analytic. Given the statistical effects noted in the last section, our purpose here is to briefly discuss two dominant patterns seen in real-life analysis from the perspectives of sample and event space.

1.2.4.1 Patterns Associated with Limiting Behavior

The previous section discussed the fact that when program values depending on (varying with respect to) different dimensions of program sample space are combined together, they lead to very distinctive patterns in both sample and event spaces. Viewed from sample space, such combinations result in another value varying with respect to both dimensions, and containing the results of combining all possible pairs of values from the event spaces of each of the two values. The event space manifestations of the same combination are even more distinctive: As hinted at by the equations in the last section, the event space of the resulting value is created by from a process conceptually similar to *convolution*.

While the patterns emerging from a particular combination of values can be distinctive, the event space patterns that result from a *series* of such combinations are more notable still. Figure 183 shows the step-by-step results of performing successive summations of independent program values in the event space; the behavior illustrates a classic case of convergent behavior associated with the central limit theorem [Drake 1967]. Convergence is not limited to the case of value summations, however: Figure 184 shows reminiscent behavior in the context of multiplication operators, and many other combinations of values will exhibit similar stochastic convergence.

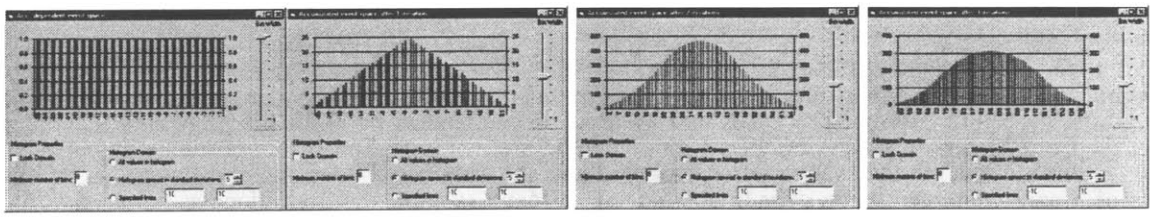

Figure 183: Statistical Patterns Emerging from Additions of Independent Values in a Loop.

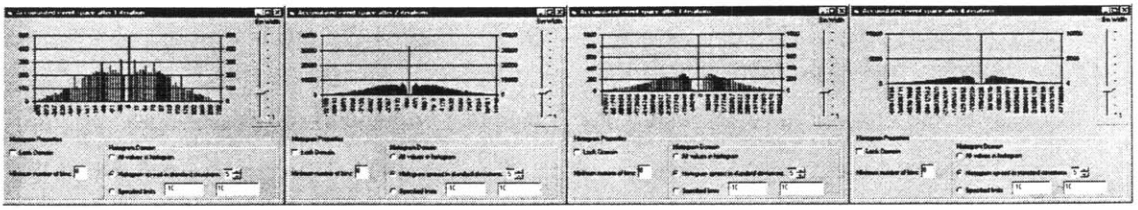

Figure 184: Statistical Patterns Emerging from Multiplications of Independent Values in a Loop

It should be emphasized that the convergence effects seen here are not limited to combinations between completely independent values. Rather, they reflect variants of the convergence behavior associated with the central limit theorem, which applies in a surprisingly wide variety of contexts [Drake 1967] and occurs for a broad range of conditions.

The presence of limiting behavior of this sort is extremely important from the point of view of the information associated with program quantities. In particular, it might be naïvely expected that as a program quantity is successively combined with unrelated program quantities, it would grow increasingly difficult to characterize. Indeed, we would expect that the number of distinct values represented in the sample space of that quantity would increase geometrically with the number of values combined, and that the event space value would therefore grow in width and complexity. The analysis of the previous section and the diagrams illustrated above suggest that this is not in general true: While the dynamic size of a program quantity's sample space does indeed grow with each successive value that is combined, this causes a progressive simplification of the quantity's event space on account of statistical effects. After a moderate to large number of such combinations, the vast bulk of the event space can be quite narrowly confined relative to its total range of variation.

1.2.4.2 Patterns Associated with Value Dependence

The discussion above focused on limiting behavior that arises when combining values that exhibit relatively low amounts of statistical dependence. Such behavior leads to rich, predictable patterns in the event space of program values. But frequently computational systems are characterized not so much by the independence exhibited between values, but by strong dependencies connecting program quantities.

In programs modeling external processes and systems, dependencies between program values frequently reflect feedback effects in the system being modeled and are manifested in strong correlations between the results of a particular calculation over time. Indeed, in many systems the dependencies between variables

represent the greatest barrier to effective system analysis by means of classical statistical techniques. [Taylor 1995]

In other programs, control flow dependencies play a large role both in shaping program behavior. Just as inter-value dependencies can prevent the effective application of traditional analysis methods for modeling applications, they can serve to thwart traditional analyses of other classes of programs by obscuring the path of runtime execution.

Finally, programs performing large amounts of calculation frequently exhibit large amounts of inter-value dependency, with the values throughout the calculation strongly dependent on the seed values and on one another.

The patterns that emerge on account of dependence between program values are less easy to characterize than patterns arising from program value independence. (For example, program values can exhibit high dependence while completely correlated or completely anticorrelated). While they are less easy to directly recognize, dependence effects are among some of the most important factors in shaping program behavior. The sample space approximating value domain discussed in Chapter 11 is motivated in large part by the need to effectively model value dependence and convergence effects in order to accurately understand and approximate run-time behavior.

1.3 Uncertainty during Program Analysis

The previous sections have been limited to describing the nature of uncertainty concerning program execution context, and ways of looking at program operation in the presence of that uncertainty. The discussion provided a simple probabilistic characterization of implications of uncertainty in a program's execution context. This was *inherent* uncertainty concerning the details of program operation that grew out of a stochastic system, independent of any observer.

This section represents a fundamental shift in emphasis. Here we are moving from a discussion of *what obtains during program operation* to a discussion of *what analysis can learn about what obtains during program operation.* Rather than focusing on the character and consequences of *a priori* uncertainty that is inherent in a stochastic system of the sort that we have outlined, our interest here turns to epistemology. In particular, we are interested in understanding what and how much program analysis can learn about the behavior of a stochastic system while still guaranteeing a widely desired characteristic of analysis implementation – analysis termination.

1.3.1 Guaranteeing Timely Termination

Existing program analyses vary widely in character, but for the sake of this discussion they can be clustered into two broad categories: Analyses that operate at run-time on precise knowledge of their inputs, and those that operate on partial or (far more typically) no information concerning the input to the program. These two forms of analysis differ widely in a number of areas, but one of the most pronounced of these differences lies in the guarantees offered with respect to analysis running time and termination.

Because run-time analyses operate by "piggybacking" on top of a standard analysis, the termination of such analysis is typically tied directly to the run-time of the program being analyzed. By contrast, the termination of static analysis algorithms is typically guaranteed. Indeed, run-time analyses notwithstanding, frequently strong termination guarantees are regarded as a prerequisite for conducting realistic analysis. Motivated by TACHYON's general philosophy of promoting user control over the quality and running time of program analyses, we would like to provide the user with the option of guaranteed analysis termination. Unfortunately, such termination guarantees do not come for free – they require potential compromises in the quality of the analysis performed. In particular, the ability to either prematurely truncate a path of analysis or to summarize the effects of an arbitrarily long segment of an analysis (such as a loop or procedure call) requires the capacity to model the unknown effects of execution along the segment of the path that is not explicitly simulated. At a level of analysis *implementation*, therefore, the capacity to ensure termination leads directly to the need to represent unknown values within a particular execution context. At a conceptual level, the desire to guarantee termination greatly complicates the effects of uncertainty that we examined in the last section. In particular, uncertainty regarding program values during analysis can now emerge either directly from uncertainty regarding program inputs, or indirectly as a result of calculating conservative approximations to segments of execution.

1.3.2 Summarizing Multiple Executions at Program Points

A closely related additional source of uncertainty in the output of many analysis algorithms emerges from the desire to summarize aspects of program execution with respect to points in the (original) program source code. A given point in the source code of the program may be executed many times for a particular execution context. As a result, such summaries inevitably involve uncertainty that arises from trying to summarize properties of program execution at the same program point over the more than one point in the execution of the program.

It is important to recognize that this source of uncertainty – while an important one in practice – is distinguished in an important manner from the uncertainty that results from the need to guarantee termination. In particular the desire to summarize output information over program points does not

necessarily degrade the quality of the information maintained *during* analysis – assuming termination of the analysis is guaranteed, one could delay the "summarization" (more technically, "abstracting" by taking the least upper bound) of the collected information until output. Most implementations of analysis algorithms do take the approach of abstracting during the analysis algorithm itself, based on pragmatic consideration of the running time of the analysis. (Recall from Chapter 2 that while the fundamental approximativity of the "composition of abstractions" and the "abstraction of compositions" [Ayers 1993] is guaranteed by the Kleene Fixed Point Theorem, the "composition of abstractions" is likely to less accurate but quite a bit faster than its counterpart.)

1.3.3 Summary

This section has briefly surveyed two of the issues that combine to limit the practical accuracy of analysis in uncovering the dynamic characteristics of run-time behavior that were described in Section 1.2 of Appendix 1. In particular, while run-time analysis systems could in principle simulate program evolution in different execution contexts using only concrete values and the distinguished $\perp$ element, an additional refinement is needed for an analysis system that wishes to achieve timely analysis termination and to summarize analysis results across program points. In particular, while the patterns of data flow during analysis may closely resemble those that obtain during program execution, such an analysis system must also be capable of modeling "unknown" ($\top$) values. The presence of the top value in analysis dynamics fosters efficiency but can exact high cost in terms of analysis precision.

1.4 Conclusion

This chapter has summarized factors influencing the quality of the information manipulated during analysis. The first section of the chapter established a framework for thinking about the consequences of uncertainty with regards to program input. In particular, the chapter presented a model a program is viewed as stochastically executing the program on a sample space of possible execution contexts. Within this framework, program inputs and internal program values represent random variables defined over the sample space of execution contexts. Following the establishment of this framework, attention focused on understanding how the random variables associated with program values (input or otherwise) combined together to yield a resulting program value. We characterized this interaction from both a sample space perspective and from the somewhat more familiar but less fundamental viewpoint of event space. This discussion helped us to understand the character of the uncertainty concerning aspects of program operation that inherently arises from uncertainty concerning the program execution context, independent of any analysis.

Following the discussion of the character and implications of uncertainty in program execution, we continued on to briefly survey the issues surrounding *analysis* in the presence of uncertainty regarding program input. While the previous section focused on the nature of uncertainty and its implications independent of an observer, the discussion of analysis dealt with issues regarding epistemology – what knowledge it is feasible to obtain, given pragmatic constraints such as the need to guarantee analysis termination and the desire to summarize analysis information applying to particular points within the source program. In order to accommodate such constraints, we were required to broaden our model of execution to include the unknown ($\top$) element.

It should be stressed that our attention in this chapter has remained fixed on constraints that impose *upper bounds* on the quality of information manipulated during analysis. Frequently additional dilutions of analysis information are viewed as *desirable* for the reporting of analysis information or to guarantee certain running time for the analysis algorithms. Discussion of pragmatic tradeoffs of this sort are beyond the scope of this chapter, but are touched on in Chapter 12 from the perspective of a particular analysis implementation.

This chapter forms the conceptual basis for the "sample space representation" detailed in Chapter 11, and the design of that abstraction closely adheres to the conceptual framework presented here.

Appendix 2 Brief Reviews of Extant Representational Schemes

This appendix surveys a number of value representations found in the analysis literature. While extensive work has been done on representations for pointers, the focus here is on representations for *scalar* values. We are particularly interested in representations that permit expressing information on *partially known* values. We systematically compare the representations with respect to nine criteria (each associated with a row in Table 38).

	3-Level	Set-Based	Interval	Predicate-Based
Representational Breadth				
Compactness				
Cost of Value Operations				
Abstract Domain Height				
Epistemic Transparency				Depends
Closure under Operation				Depends
Scalable Analysis Precision				
Sensitivity to Frequencies	N/A			
Capacity to Maintain Contextual Dependency	N/A			

Table 38: Table Comparing the Strengths of Different Possible Value Representations

2.1 The 3-Level Representation

The placement of the 3-level representation within the first column of Table 38 reflects the fact that it is by far the most widely used of the representations examined. Program analysis tools running the gamut from the most basic of constant propagators [Aho, Sethi et al. 1986] to the most sophisticated of abstract interpretation systems [Chen 1994] use this representation to capture the values of program values. The representation consists of a 3-leveled lattice, as shown in Figure 185. Program values can either be located

at the top of the lattice and completely unknown ($\top$), known to be a particular concrete value (represented by the middle tier) or no value at all ($\bot$).

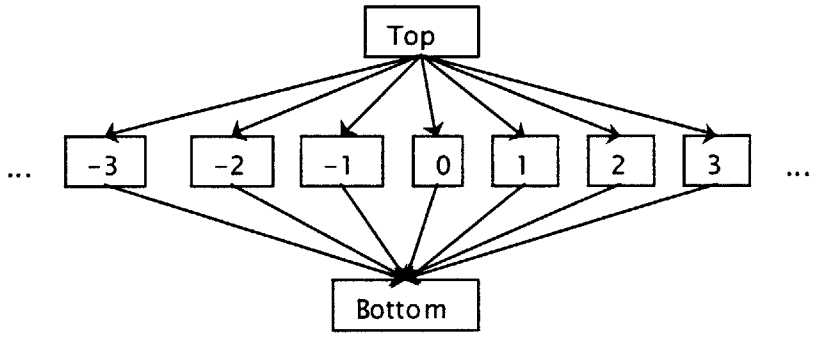

Figure 185: The Lattice Associated with the Three Level Representation

Unlike the interval representation to be presented in Section 2.2, the lattice presupposes no ordering between values, and can be used to capture any type of value – pointers, numeric quantities, characters and strings, etc. Such abstract values require marginally more space to represent than is required by the value itself (or 1 additional bit)[81].

The cost of operations on 3-level values is slightly greater than would be required for performing such operations on standard semantics values from the same domain, but still extremely low. For certain high-cost operations, the abstract analogue to an operation may even be less expensive due to the fact that the operation only needs to be executed for cases where all operands are known. Moreover, the fact that the value is of height two allows for rapid convergence of loops during abstract execution.

As attested by its widespread use in analysis tutorials [Chen 1994] [Aho, Sethi et al. 1986], the abstract domain is easy to conceptualize and is without question closed under typical value operations.

The primary shortcomings of the 3-level representation stem from the domain's inability to scale to more precise analysis. Although one could in principle attain more *rapid* analysis by eliminating part or all of the middle tier of the abstract domain (as done in [Ayers 1993]), there is no way to "tune" 3-level values so as

[81] By selectively eliminating certain values from the concrete middle tier, the entire abstract value can be captured within the space that would be required to represent the concrete value. [Ayers 1993] makes use of a strategy for representing constants that might be applicable in this context as well.

to increase their representational capacity. For many analyses this is not a problem, but offering greater choice to the user (in this case, the developer) is in general desirable.

Because the 3-level domain has no capacity to represent values about which the analysis yields *partial* but incomplete knowledge, the issues of maintaining frequency estimates or contextual dependency information within partially known values does not arise.

In general, the 3-level domain is well-suited to a wide variety of analyses, but cannot provide the precision required by other collecting domains.

2.2 Intervals

The second most popular representation for values among those shown is the interval representation. The most important manner in which this abstraction differs from the 3-level value representation is in the fact that it moves beyond the fully known/completely unknown dichotomy for representing values, and offers a means of approximating *partially* known values. The use of the interval representation and its associated algebra has been extensively studied, both as a general purpose computational technique for use in mathematical programs [Nickel 1986] and as a representation for use in program analysis. Unfortunately for our purposes, while many of the results regarding interval arithmetic are useful for constructing computational models of mathematical systems that employ interval approximations to quantities (e.g. to represent error bounds), they have little or no applicability to general-purpose code. In particular, the use of such results typically requires *a priori* analytic knowledge of aspects of the mathematical system being modeled, such as the derivatives of formulas being evaluated within the code. [Nickel 1986]

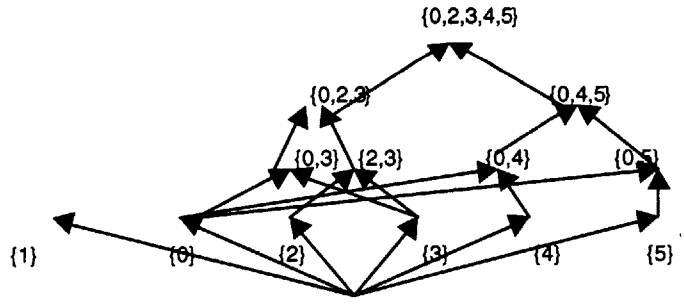

Figure 186: The Lattice Structure for the Interval Representation.

Nonetheless, previous analysis systems have made strong use of intervals during program analysis. [Weise and Ruf 1990; Osgood 1993] This is hardly surprising: Intervals capture the possible range of uncertainty

concerning a value in a compact, intuitive and efficient manner. Unfortunately, they also suffer from some serious shortcomings within program analysis. We will briefly discuss how such representations measure up according to the standards laid out in Table 38.

2.2.1 Type Restrictions

The first significant shortcoming of intervals is the assumption of an ordering relation among elements of the standard semantics types being represented. An ordering is naturally present for numeric types such as integers and floating point numbers; even types such as bytes or UNICODE characters can be reasonably coerced into an ordering system. The notion of an ordering is less meaningful in the context of pointer types. In particular, while we may speak of two pointers that share the same allocated region being ordered with respect to one another, such a concept is of dubious value when applied to pointers drawn from different states or different allocated areas within the same state (*e.g.* from the stack and from the heap). For a String type, an ordering may be imposed, but demonstrating code in which representation of a string interval offers useful results requires imagination indeed. Finally, types such as *Boolean* exhibit no natural ordering, and cannot be fit comfortably into an interval framework.

2.2.2 Analysis Efficiency: Strengths and Weaknesses

Proceeding on in our list of criteria in Table 38, two of the strongest advantages of the interval representation relative to other partially-known value abstractions are its minimal direct computational resource demands. In particular, the interval representation is very small (requiring just two concrete semantics values to represent, even allowing for encoding of the distinguished $\top$ and $\bot$ variants), and admits to very efficient naïve implementations of most operations.

Nonetheless, analyses involving the representation can be very inefficient, depending on what decisions made with regards to enforcing height restrictions on the abstract domain. Recall from Section 7.4 that the speed of analysis is in large part determined by the speed with which program loops can be iterated to a fixed point. The amount of time required for such convergence can depend strongly, in turn, on the height of (i.e. the length of the longest chain in) the abstract value domain. In order to achieve such convergence in a timely manner while using the interval representation, the height of the value domain must be capped in an effective – and hopefully natural – manner. While it is tempting to always treat $[a,b] \sqcup [c,d]$ as $[\min(a,c), \max(c,d)]$, this strategy is inadequate here: The height of the resulting domain is simply too great. To see this, the reader is referred to Figure 187. In order to analyze the effects of the loop, the system must the iterate the loop until the loop entry state has converged (see Chapter 7 and Chapter 2.) For each iteration of the loop, the analysis system will take the least upper bound of the new and accumulated loop entry state.

For an abstract state domain of finite height, this process will eventually result in no changes. Certainly, the loop convergence can occur no earlier than the point at which the estimate of the value of the loop iteration variable *i* reaches a fixed point. (If the estimate of *i* is still changing, the estimate of the loop entry state will be as well.) But if the value of *i* is approximated by an interval and we use the least upper bound strategy above, the progression of the values of *i* will be as shown in Figure 188, or even longer (as the caption notes). The height of the abstract domain here is one more than the number of distinct integers – a large number indeed! Clearly, we need an abstract domain for values which does not permit values to climb such a long chain before converging.

```
For(i = 0; i < 1000000; i++)
    {
    …  ; // code not affecting i
    }
```

Figure 187: A simple situation in which a naïve structure for the abstract interval domain yields unacceptably long analysis time.

```
i = [0,0]
i = [0,0] ⊔ [1,1] = [0,1]
i = [0,1] ⊔ [1,2] = [0,2]
i = [0,2] ⊔ [1,3] = [0,3]
…
i = [0,999998] ⊔ [1,999999] = [0,999999]
i = [0,999999] ⊔ [1,999999] = [0,999999]// Fixed point!
```

Figure 188: The convergence sequence for the loop variable *i* within a naïve implementation of the interval domain.

Somewhat artificial strategies to limit the height of the interval domain are easy to advance. For example, [Osgood 1993] makes use of a strategy that simply limits the number of least upper bounds in which an interval can participate. Each interval carries around with it a count of the number of least upper bound operations in which it has participated. When taking a new upper bound, this value is checked; if it is equal to the maximum legal number of least upper bounds, the new value resulting from the upper bound will be $\top$. While such strategies succeed in limiting the height of the domain, they require extra mechanism and degrade the quality of the analysis that would otherwise take place. Based on these considerations, Table 38 credits the interval representation with poor enforcement of abstract domain height.

A second significant difficulty with the interval representation lies in the fact that it is not closed under all common operations. In particular, applying an operation such as division when the denominator holds a value represented in the interval representation may yield a result that cannot be expressed with that

representation, and must be approximated by $\top$[82]. In particular, if the denominator of the expression crosses "0", the values resulting from the expression can fall in either of *two* very broad intervals. In the event that the numerator is also represented by an interval that crosses zero, the values resulting from the expression can potentially fall anywhere along the real line. Table 39 illustrates these different possible results, highlighting those cases in which the interval loses all precision. Although it is technically possible to approximate the results shown in the shaded regions with the trivial interval $(-\infty,\infty)$, this voids a major purpose of maintaining the interval representation (namely, to effectively bound the analysis results) and adversely impacts the precision of all downstream analysis. So although the interval representation is closed in theory under operations such as division, from a practical standpoint of performing effective, high-quality analysis it is *not* closed under such operations.

Value of (a,b)	**Value of (c,d)**	**Span of Possible Value for (a,b)/(c,d)**
<0	<0	(a/d, b/c)
$a<0<b$	<0	(b/c, a/c)
>0	<0	(b/c, a/d)
<0	$c<0<d$	$(-\infty,b/d) \cup (a/c,\infty)$
$a<0<b$	$c<0<d$	$(-\infty,\infty) \equiv \top$
>0	$c<0<d$	$(-\infty,a/c) \cup (a/d,\infty)$
<0	>0	(b/c, a/d)
$a<0<b$	>0	(a/c, b/c)
>0	>0	(a/d, b/c)

Table 39: Regions in which the Quotient of an Interval Division can Lie.

In contrast to the 3-level representation (which is limited to the representation of values as completely known or entirely unknown quantities), the interval representation captures information regarding partially known quantities. Unfortunately, like the 3-value domain examined in the previous section, the amount of information stored in the interval representation is fixed -- the representation has no capacity to scale so as to perform more precise analysis over an entire program or components thereof.

The representation of quantities by the interval representation involves an estimate of the possible set of concrete values that can be taken on by that quantity. Given that there is a *set* of concrete values possible, two questions immediately arise:

- What is the relative *frequency* with which the different estimated values for the quantity obtain?

[82] Note that within the interval representation, $\top$ can be represented by a trivial element representing the entire span of the appropriate data type that is drawn from the interval representation.

- Given a combination with another quantity represented in the same manner, what are the different pairings of values from each value that are possible?

These questions form the basis for table rows 8 and 9. Unfortunately, the interval representation provides very weak – and even inconsistent – answers in both of these questions. The next two sections investigate these issues in greater detail.

2.2.3 Relative Frequency

The representation of a value v using an interval representation [a,b] states that the value of v is guaranteed to fall within the interval [a,b], but makes no additional statement as to the relative likelihood that v will fall into any particular subinterval of [a,b]. Lacking any additional special knowledge concerning the system at hand, it is tempting to make the "simplest" assumption : To wit, that v has uniform probability of being associated with any particular element of [a,b]. But statistics has taught us that such a priori assumptions are suspect indeed (for example, it may be "more natural" to assume that v has a uniform probability density over the real interval [a,b] – an assumption that has rather different consequences for non-uniformly spaced floating point values.)

The association of a uniform probability distribution with interval representations is also inconsistent, in that the probability distributions associated with the results of applying operators to with uniform distributions are typically far from uniform – an aspect of program behavior that was touched on in 11.2. Figure 190 and Figure 191 show distributions induced by applying different common binary operators to uniformly distributed values. It can be readily appreciated that the results are not typically described by a uniform probability distribution, and that very distinctive patterns in the probability distribution of values can arise through the repeated combination of values (a fact discussed in greater detail in Section 1.2 of Appendix 1) . The assumption of uniform distributions in values is unrealistic, and can prevent recognition of important dynamic patterns.

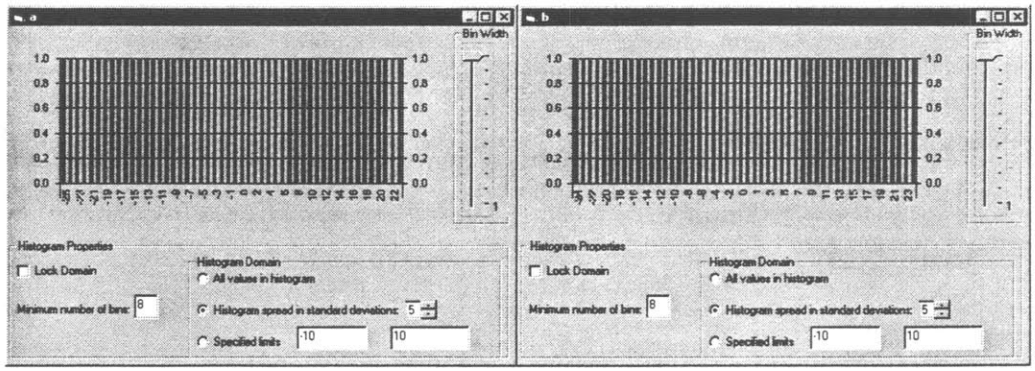

Figure 189: Uniform Event Space of (Independent) Variables *a* and *b*

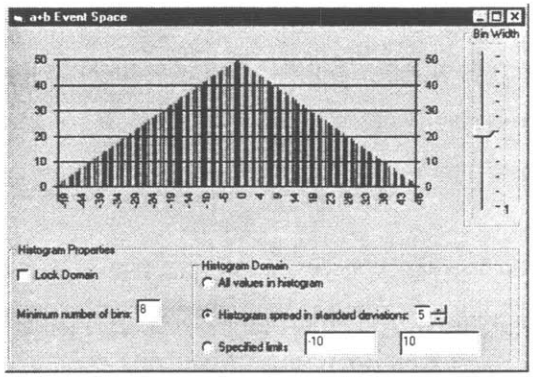

Figure 190: Non-Uniform Event Space of *a+b*

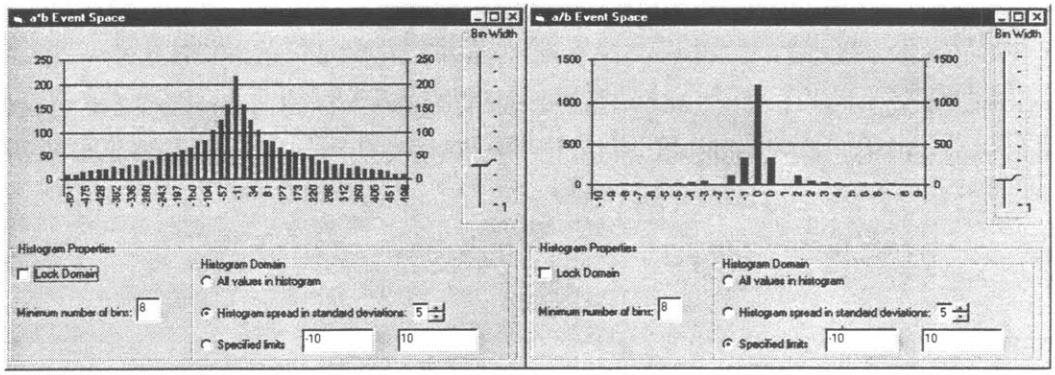

Figure 191: Non-Uniform Event Spaces of *a*b* and *a/b*

Given the commitment to representing partially known values, strong arguments can be made for the desirability of attempting to capture the relative frequency of the possible values. (See Section 2.2.3.) But this extra information is only meaningful in a system that maintains information concerning the contexts in which different possible values apply. It is this variety of information that we turn to next.

2.2.4 Contextual Information and the Loss of Information

As was discussed in Appendix 1, program values can be characterized as random variables defined over an event space whose configurations are individual possible execution contexts. Section 1.2.4.2 of Appendix 1 noted the critical role that the statistical dependencies between such values play in shaping the course of program execution. For example, many pieces of numerical code are designed to model systems containing a large amount of feedback; on account of this feedback, many values within simulations of such systems exhibit high correlation. [Taylor 1995] In other systems, it is very typical that the inputs to the program will exhibit high degrees of correlation, and that the operations encountered during program flow lead to myriad low-level dependencies.

Because program dependencies influence program behavior so strongly; modeling such dependencies is strongly desirable for an analysis that wishes to tightly approximate either the path of program execution or the range of concrete values taken on by program quantities during analysis. The interval representation has no capacity to model such dependencies – a shortcoming that relegates it to performing extremely inaccurate analyses in many circumstances.

For an example of a trivial but not implausible shortcoming that may be encountered during analysis, consider an expression that computes the square of a double precision floating point value:

```
x*x
```

Now consider the analysis of this code when x is associated with the interval value [-1,1]. Because an interval-based analysis system has no capacity to recognize the inherent (trivial) dependence between the operands on each side of the "*" , the system will apply the general rule for the combination of [a,b]*[c,d]= [min(a*c,b*c,a*d,b*d), max(a*c,b*c,a*d,b*d)]. As a result, the combination will return the interval value [-1,1] – a gross overestimate of the actual interval in which the value could fall ([0,1]). Figure 192 illustrates the substantial gap between the estimate given by the interval representation and the actual distribution that results from the combination.

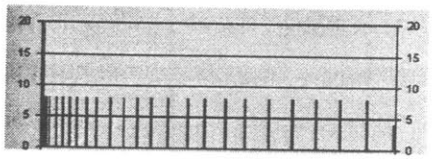

Figure 192: Distribution Resulting from Multiplication of a Value
Associated with Uniform Probability over the Interval [-1,1] Times Itself.

When several quantities are combined (e.g. using an operator such as "*" above), the interval representation naively assumes that *all* combinations of concrete values from each of the quantities are possible. This leads to a very wide – and overly conservative – estimate of the interval in which results can lie. By contrast, the presence of dependence information provides additional constraints on what concrete values in each quantity can be combined with what concrete values in the other quantities, and thus provides a tighter bound on the possible interval in which the result of an operation can be found.

While the example being manipulated here is somewhat artificial, it illustrates the root cause of a critical shortcoming of interval representations – their tendency to rapidly lose accuracy when combined with other values. [Osgood 1993] This predisposition is manifested in any expression in which the subparts of the expression exhibit dependence. Some examples of this are shown in Figure 193, Figure 194, Figure 195, in which the actual bounds tend to be significantly – and sometimes drastically -- tighter than those estimated using interval representations, or with the assumption of independence.

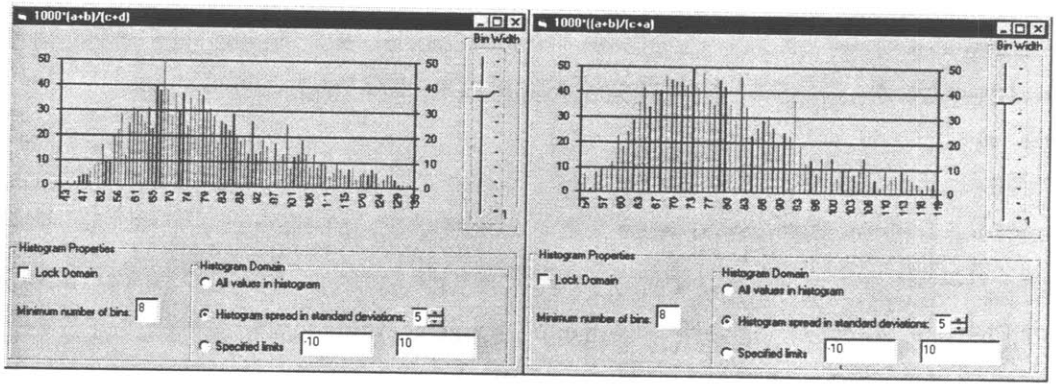

Figure 193: Comparisons of the Results of Two Expressions with
Dependent and Independent Parts

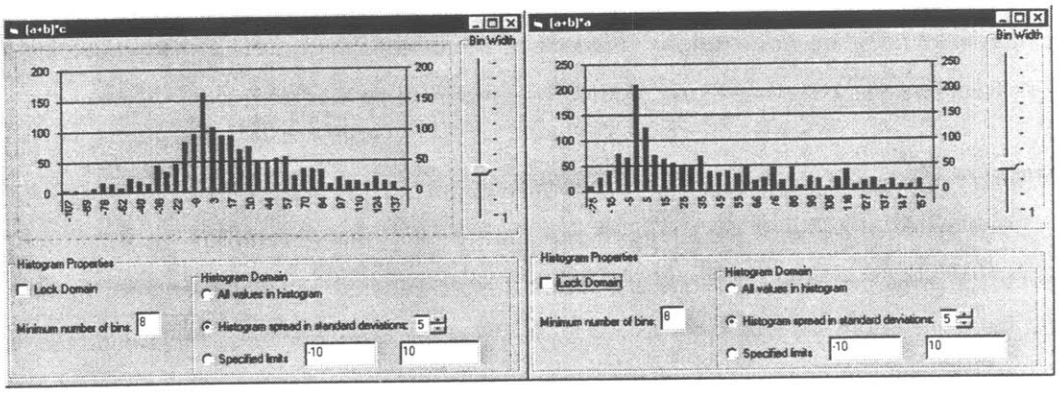

Figure 194: Comparison of Accumulate and Multiply Expression for Independent and Dependent Values

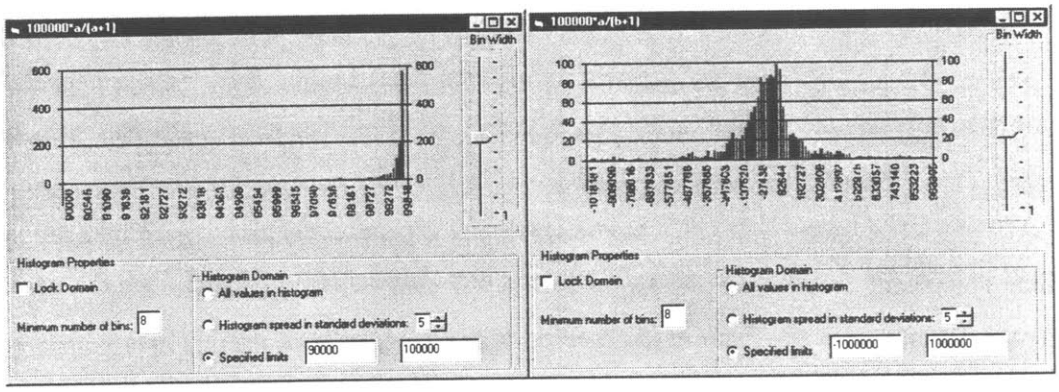

Figure 195: Comparison of the Event Space Resulting from Two Accumulate And Divide Expressions with Different Levels of Internal Dependency

This phenomenon is especially pronounced – and particularly important – in the context of *repeated* combination of values, as occurs in program loops. Because the values undergoing repeated combination on one iteration are frequently derived from those manipulated in the previous iteration, there is a great deal of natural dependence exhibited between iterations. This can result in much tighter possible bounds than would be naïvely assumed. When the code is simulated using values approximated with the interval representation, such values tend to diverge rapidly during analysis. This divergence arises from the fact that the current values in each iteration are combined in a grossly conservative manner with the values of the previous iteration, yielding an even more exaggerated result.

2.2.5 *Lack of Scalability*

An important shortcoming of the interval representation is the lack of *scalability* of the precision with which the representation can capture information. This lack of flexibility forces user-defined collecting analyses seeking high-precision analysis to make use of another approximating domain representation.

2.2.6 *Summary of Interval Representation Tradeoffs*

This section has surveyed some of the strengths and shortcomings that accompany the use of the interval representation for program values. While the interval representation is compact, intuitive, and relative efficient for most operations, its simplicity masks some important shortcomings. Most importantly for our purposes, it has no capacity to scale to higher precision when this is needed and appropriate computational resources are available. Moreover, the information it maintains concerning partially known values is frequently insufficiently general and precise to maintain tight approximations to the actual range of possible variation in the context of following combination with other values. The representation is not in general effectively closed under operations such as division – division by any interval that spans zero yields a range of infinite extent. Because of failure to maintain information related to the *dependencies* between values, the representation requires making the "worst case assumption" of independence when two or more values are combined. This frequently yields approximations to the resulting interval that grossly overestimate the actual range in which the value could fall. This shortcoming is particularly pronounced in cases where the code exhibits significant amounts of dependency between values, and frequently renders the interval approximations virtually useless. Even in those cases where the calculated interval approximation tightly bounds the true range of potential variation, the failure to model the probability distribution of values over that interval can mask important patterns (*e.g.* the recognition that the result of interest is almost certain to lie within a certain narrow range of the nominally possible interval due to statistical effects).

Finally, while the model admits to low-cost combination with other values, by itself it offers no natural limit to the height of the abstract value domain. As a result, while particular operations involving such values may complete quickly, an analysis involving interval values may require an extended period of time to converge – a tendency complicated by the above-mentioned predisposition towards strongly overestimating the possible range of variation of the value. This problem can be (somewhat artificially) fixed by imposing a particular maximum height on the domain [Osgood 1993], but such a change is artificial and somewhat unsatisfying.

The interval representation is popular, easy to implement, and addresses a very real need — the desire to represent *partially known* values. Unfortunately, it is associated with substantial costs in analysis speed, and

such a strong tendency to lose precision that it severely undercuts the primary motivation for the adoption of a partially known value representation.

2.3 Set-Based Representation

2.3.1 Overview

We turn now to a discussion of a third means of approximating scalar values. We will term this approximation the "set representation" for the remainder of this document. Like the interval value representation, this technique rejects the traditional dichotomy of values into "completely known" and "completely unknown" categories, and allows the representation of *partially known* values. Unlike the interval representation, however, the set abstraction serves as an *extensible* representation of partially known values. While a non-extensible representation such as the interval representation maintains the same character and size when representing different values, an extensible representation can change the amount and quantity of information maintained between different values in order to capture information with added precision.

The set representation describes the value of a program quantity as one of an enumerated *set* of particular concrete values. For example, an integer value that was known to be greater than zero and less than or equal to five would be represented as { 1, 2, 3, 4, 5 }. A floating point value known to be either identically zero or identically one would be associated with the set { 0.0, 1.0 }. This domain forms a natural lattice, as shown in Figure 196. Above the bottom element $\perp_T$, the lowest layer of the domain of type T is composed of singleton sets of values; values associated with such sets are precisely known. Each successive layer of the domain is associated with larger and larger sets, representing approximations to two or more members of lower layers. Finally, the domain is associated with a distinguished element $\top_T$, which signifies *any* possible set of values. As discussed in Section 2.3.5.4,this domain must be of finite height in order to guarantee convergence of the abstract interpretation. The set representation assumed a very important role in [Osgood 1993] Variants of the set representation have played important roles in the analysis of pointer structures. [Jones and Muchnick 1981].

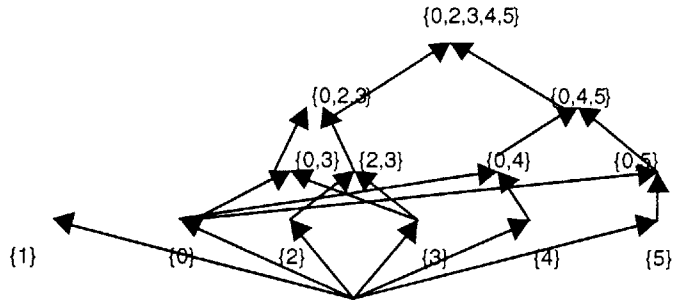

Figure 196: A Depiction of the Lattice Associated with the Set Domain

Because set can be composed of any type of value, the use of the set representation is not restricted to any particular value types. By contrast to the interval representation (which only applies to ordinal types, and is an awkward fit even for some such types), the set representation is well-suited to representing values from domains as diverse as strings, bytes, floating point values, and pointers.

The space consumption of a member of the set representation can vary widely, depending on the size of the set (and thus the height of the value in the lattice). While the best-case cost of a value represented using such a scheme is relatively low, the space required to represent the value can grow very large for values that are less precisely known.

Given arguments $a_0 \ldots a_{n-1} \in \wp(\text{Value}^{\#})$,

$\oplus(a_0, a_1, \ldots, a_{n-1}) = \text{Truncate}(\cup\{\oplus(v_0, v_{1,\ldots,} v_{n-1}\} \mid (v_0, v_{1,\ldots,} v_{n-1}) \in (a_0 \times a_1 \times \ldots \times a_{n-1})))$

Where Truncate : $\wp(\text{Value}^{\#}) \rightarrow \wp(\text{Value}^{\#})$ is defined as

$\text{Truncate}(S) \equiv \{\ S \text{ if } |S| > \text{the maximum allowable domain height}, \top \text{ otherwise } \}$

Figure 197: Combinatorial Character of Value Combination in the Set-based Semantics.

In a similar manner, the cost of operations performed on members of the set representation can vary widely. In the context of a deep lattice, the costs of such operations can be *very* high. As suggested by the characterization of value combination in Figure 197, an operation combining a set-value of size a with another value of size b requires time $\theta(ab)$, as each value from one set will need to be combined with *every* possible value from the other set. This imposes a very high overhead on maintaining less precisely known values. The superlinear scaling of this overhead alone militates for the adoption of a low-height abstract domain. In this light, it is worth recalling noting that in representations of partially known values, the height of values within the abstract domain can be increased not only at the points where least upper bounds are

taken, but also by general-purpose value combination operators. In other words, combining a sequence of operand values of maximum height n can yield a resulting value of height $>n$ (and sometimes $\gg n$). For example, Figure 198 shows two set-based values of cardinality two being combined to yield a value of cardinality 3. That the resulting value is located higher in the abstract domain than either of the operands can be readily appreciated by noting that it is a legitimate approximation to either value (that is, both $\{ 0,1 \} \sqsubset \{ 0, 1, 2 \}$ and $\{ 1, 2 \} \sqsubset \{ 0, 1, 2 \}$). In fact, the result also happens to be the least upper bound of the two operands. Convergence of the analysis would still be guaranteed if we were to rely upon least upper bound operators in analysis to enforce the limit on the height of elements drawn from the abstract domain (i.e. to truncate values exceeding the height of the domain to $\top$) [Osgood 1993]. Given the high cost of manipulating large set-based values, however, it is prudent to perform the extra checks necessary to immediately convert any excessively large values returned by an operator directly to $\top$ within the value combination machinery. This limit is enforced in Figure 197 through the use of the *Truncate* function.

```
{ 0,1 } * { 1, 2 } = { 0, 1, 2 }
```

Figure 198: Combination of Two Partially Unknown Values Yields Value Higher in Lattice.

While the height of the value domain provides a cap on the maximum size of the values that must be combined during abstract execution, it also imposes an upper limit on the time required for the convergence of fixed-point iterations conducted during analysis. (See Section 2.3.5.4). Given a "deep" abstract domain that permits large sets for value approximation, the convergence will require additional time.[83]

The set abstraction for partially known values is a very intuitive representation for all language types, even more so than the interval representation (which has less clear semantics for types that are unordered or admit to many possible orderings).

The set representation is also unlike the interval representation in that it has the virtue of being closed (in practice as well as theoretically) under any construct: Apply any operator to one or more sets of values, and it is possible to describe the resulting values as a set. We can describe this process in a more precise manner below:

That is, the result of applying an operator to a sequence of values drawn from the set representation is just the union of singleton sets that record the application of that operator to every possible combination of

values drawn from the operands. If the full set of possible resulting values is too large to fit within the lattice, it is approximated by $\top$. Thus, aside from domain height constraints that apply to all representations of partially known values, the set representation is closed under combination using any operator.

The set-based representation provides an eminently simple means of increasing or decreasing analysis strength by increasing or decreasing the maximum set size (thus directly establishing the height of the domain). Increasing domain height allows for more exact representation of less precisely known values, while decreasing domain height will approximate such values by the compact but impoverished value $\top$. [Osgood 1993] relied heavily on this technique to provide user-customizable analysis precision, with considerable success for small programs. This mechanism is indeed convenient and effective as a way of trading off analysis time for accuracy: Adjusting domain height controls both the time required to perform value combinations and the upper bound (as well as expected) time required for fixed-point convergence during abstract execution. While having a single parameter available to control both mechanisms can be *convenient*, it does lack some flexibility. We would ideally like a representation, which allowed *separate* control of the two sources of additional analysis time (the time taken by individual operations *vs.* the time required for loops to converge);

The effective use of domain height as a way of adjusting the precision of the set representation is limited by the poor scaling of the analysis time. In particular, because the combination of multiple values using the representation increases in a superlinear fashion with the heights of the values in the domain, it is not realistic to make use of a broad range of domain heights.

The set-based representation is designed to capture information regarding *all* possible concrete values that may be associated with a particular program value. This is valuable information indeed: It allows the collecting domain access to precise information regarding all possible values associated with a program value with less "blurring" than occurs with a non-extensible representation such as the 3-tier or interval abstraction. Given the size and computational resource demands that can be associated with the set-based representation, it is natural to hope that the representation would be strong enough to permit tightly bounding the results of operations performed on values maintained using the representation. Unfortunately, this is not the case, and for much the same reasons as discussed for the interval representation in Sections

[83] Note that the detailed scaling of the convergence time with the height of the domain depends on the collecting domains and abstract analysis algorithms employed – see 7.4.

2.2.3 and 2.2.4: The representation's precision is impaired by the lack of *frequency* and *inter-value dependency* information.

As discussed in Section 1.2 of Appendix 1, typically code is associated with statistical effects that effectively bound the likely ranges taken on by a value to a smaller interval than might naïvely be expected. Frequency information is thus critical for accurately predicting the patterns of data associated with particular values in the program code. Such information may be of interest to the user for the sake of understanding the software system (or a real-world situation that it models), or used to perform more effective program optimization (e.g. in order to take advantage of certain cases viewed as especially likely). Although such information does not directly permit the tighter bounding of the *possible* values that result from a set of combination, it can allow for a *practical* bounding of such information. In particular, frequency information can help delineate the ranges in which the vast majority of encountered values may be found, and those ranges that are in theory possible, but which are rarely encountered in practice.

Inter-value dependency information is critical to precisely characterizing the results of value combinations. The absence of this information also requires longer value combination times. Without access to such information it is necessary to make the conservative assumption that values being combined are *independent*. This frequently leads both to gross overestimates of the possible set of values taken on by the result, as well as the very expensive prospect of combining *all* possible sets of values in the operand values according to the technique shown in Figure 197. For instance, given *a* represented using the set representation as { 1, 2}, the value of *a*(a+1)* would be calculated as { 1, 2 } * { 2, 3} = { 1, 3, 4, 6 }. This is a gross overestimate of the *true* set in which the answer can lie, which is { 2, 6 }. The actual answer is both more accurate and more compact. As is the case with interval representation, the inability to effectively recognize and deal with inter-value dependencies frequently leads to tremendous dilution of value information when abstractly executing a sequence of code, as each successive operation causes greater and greater loss of precision. As might be expected, this effect is particularly notable in loops, where the contrast between the values returned by the set representation (assuming independence between values) and those returned in practice (where dependence is likely to be very pronounced) is particularly striking. [Osgood 1993]

2.3.2 Summary of Set-Based Representation Tradeoffs

The set-based representation permits broad scaling of maximal representational size and precision by adjusting the height of the abstract domain. If the value is capable of accurately modeling the information associated with a program value, it does so in a rich and detailed manner: By summarizing the precise set of concrete values that can be taken on by that value. This information is useful for modeling all language

types, but does not come cheaply: Allowing for very detailed portrayal of values leads to long convergence times, and is associated with sharp increases in the time required to combine program values within each operation. Most disappointingly, because of its failure to recognize and model inter-value dependencies, the value representation will typically rapidly lose accuracy when analyzing a sequence of code in which value dependency plays a role; this effect is particularly pronounced in the simulation of loops. As a result, even given relatively precise specification of the possible concrete values taken on by a series of program quantities, the set-based representation will typically be incapable of precisely bounding the results of executing a series of operations on those quantities. Given the high cost in computational resources that the set representation demands, such limitations are likely to rule out its effective use for as a general value representation. Nonetheless, the set representation may have a role to play in specialized situation in which the benefits of precision are deemed to be worth the high costs they entail, and where the loss of precision from value combination is not a problem. For an important example of such a specialized need, see the discussion of pointer domain implementations in Section 11.5.

2.4 Symbolic Representational Techniques

The third and final representation shown in Table 38 summarizes the tradeoffs associated with *symbolic* representations of values, such as are found in [Clarke 1981], and are peripherally discussed in [Nguyen 1989]. [Osgood 1993] also makes use of a symbolic predicate of sorts in order to capture information regarding equality of values. Because we are discussing a broad *class* of representations and one that has been little investigated, the discussion in this section is both more preliminary and vague than is the case when discussing other representations.

Symbolic representations capture program values by representing them using a set of (typically composable) predicates, such as "·≥b", "(·&256)=0", "·/d = 5*s". The exact syntax and logical scope of the predicates can vary widely, extending from simple comparisons to constants on the low end to sophisticated clauses on the high. While the tradeoffs associated with any particular representation clearly depend very critically on the language chosen to express the predicates, we can make some general statements about the use of such systems within program analysis.

Within the analysis of a typed language, it would be difficult (and somewhat unnatural) to avoid making the language of predicates itself typed – the values being referred to are drawn from the same domains as those in the language itself, and the same will likely be true for the expressions that manipulate those values. While a *particular* predicate would will likely express a relationship that only applies to a certain subset of the possible types (and quite possibly, just a single type), a predicate language can easily be defined so as to be expressive relative to *all* types.

On account of their expressiveness, their flexibility, and frequency of application, predicate-based systems can consume very significant amounts of storage space. In general, as the expressiveness of a particular predicate semantics increases, there will be more and more inter-value relationships that can be described using such a semantics, and the cost of describing those relationships and maintaining those descriptions during analysis can become increasingly onerous. At the same time, this flexibility can allow for compact, special-purpose representations of values that would otherwise consume significant space in a approximation such as the set representation. Predicate-based systems thus offer a mixed bag of advantages and disadvantages from a space resources point of view.

The cost of operations involving symbolically expressed values is typically high, but can vary widely depending on the particular language of predicates being used. As with other value representations, the cost of performing an value combination under the abstract semantics reflects the amount of time needed to derive the character of the result within the particular system of representation, given the knowledge that is available concerning the values involved. For the richest form of symbolic representation, this task could require a symbolic reasoning system similar to those involved in theorem provers or truth maintenance systems – and a completely infeasible cost. For simpler predicate languages, the cost is likely to be high but manageable.

An additional serious concern regarding analysis cost relates to the height of the value lattice (and thus to the amount of time required to reach convergence during the fixed-point iteration common throughout analysis). Unfortunately, taking the least upper bound of two or more of such values within the symbolic representation is likely to be more complicated than in any of the other representation schemes – creating a non-trivial (i.e. non-$\top$) approximation to two arbitrary value predicates could be involved indeed. Figure 199 provides an example of what least upper bounds performed within the symbolic domain might involve, and some example results that might be returned given certain input value approximations. Guaranteeing that a particular symbolic representational system with an associated least upper bound operator is of an acceptable height could require significant care. It should be stressed that the symbolic representation shares with all other *extensible* representation schemes the fact that the maximum domain height of any value can be increased not only at program join points but also at points at which values are combined (i.e. in value operators). While such action is not required for convergence, as is the case for the "set representation" it will likely prove worthwhile to consider detecting and curtailing the creation of values that are at undesirably high levels in sample space at their points of creation.

$(\cdot = A*5) \sqcup (\cdot = A*25) \Rightarrow \{ (\cdot = A*5^i) , (\cdot = A*i, i \in \aleph > 0), (\cdot \bmod 5 = 0) \}$

$(\cdot = A*5) \sqcup (\cdot > A*5) \Rightarrow \{ (\cdot \geq A*5) \}$

Figure 199: Possible Rules for the Least Upper Bound Operator in a Symbolic Abstract Domain.

Both the understandability of the predicate system and the issue of whether the chosen predicate language is sufficiently expressive to be closed under all operations depend on the particularities of the languages involved. In general, it is rather easy to structure a language such that both of these are guaranteed, or sufficiently likely to be satisfied on average. For richer predicate systems (e.g. with multiple clauses), the understandability of particularly representations can clearly suffer, due to the complexity of the predicates allowed and the limitations on users' ability to simultaneously conceptualize all of the conditions being expressed. But on account of the reasoning above, it is genuinely doubted whether such systems would realistically be constructed. Similarly, while it is certainly possible to design a symbolic abstract domain that does *not* exhibit closure, it is also possible to formulate symbolically-based representational schemes that permit syntactic expansion *ad infinitum* as values are manipulated and combined. This guarantees can guarantee closure (given an appropriately rich set of operators within the predicates), but may also complicate the analysis of code and the guarantee that the value domain is of fixed height.

Unlike the situation with interval and set-based values, the precision of an analysis involving symbolic values can be adjusted in many ways. One option would be to take the same route as did the other domains, and lift the ceiling on the abstract domain. In essence, this permits the representation of less precisely known values without entirely abandoning *all* information concerning a value. Another technique would be to curtail time spent doing symbolic reasoning, or to avoid producing new expressions for a value except in certain recognizable circumstances.

2.4.1 Complexities of Predicate Semantics

Before continuing on to discuss additional information that can be deduced from symbolic expressions, it is worth pausing to consider a barrier to symbolic reasoning. When considering the algebra of symbolic expressions, it is sometimes easy to overlook the fact that such operations describe operations on discrete values of finite size. Because of these realities, the manipulation of values may be associated with idiosyncratic behavior that does not typically impinge upon mathematical expressions drawn from idealized algebraic systems. For example, Figure 200 illustrates a fragment of code in which naïve symbolic reasoning would allow deduction of the equality of c and b at the end of the code. But considerations regarding the size and layout of program types, the (sometimes very unintuitive, albeit consistent) semantics of value operations provide a much more complicated picture, in which *c* and *b* are equal for *certain* values of *b* and *d*, and different for others. Clearly such considerations are more than "details" to be swept under

the rug; scaling back the reasoning of the analysis until it is guaranteed to be safe is not a feasible option – such an approach is unlikely to be able to deduce anything substantial from the code. Dealing safely with the obscurities of such computational systems requires that the analysis of formulate a complete semantic model of the value representation and manipulation portion of the language being modeled. While possible, this is a distinctly non-trivial task that must confront the subtleties of the IEEE floating point specification, and the language's or host processor's implementation of integer arithmetic. Such tasks are ambitious taken by themselves, but are magnified in languages that provide the programmer with a freer hand, with such constructs as unions, untyped heaps, or pointers. Reasoning accurately and productively within a symbolic value domain clearly requires a sophisticated and complex set of tools. To do so in a manner that is cost-effective may prove infeasible.

```
int a,b,c,d
…
a = b / d
…                  // does not modify a or d
c = a * d   // while c = (b / d) * d, does c = b?
```

Figure 200: Example of Situation where Naïve Symbolic Reasoning Breaks Down.

2.4.2 *Inter-value Dependencies and Frequency Information*

Despite the capacity of symbolic values to capture a wide variety of knowledge regarding program values, there are certain pieces of information that the representation does not *directly* capture. However, the representation that is used may be sufficiently powerful to allow the deduction of such information from the symbolic expression that is maintained.

The first such set of information is that related to value dependency. Section 1.2 of Appendix 1 detailed the critical importance of inter-value dependencies in determining the character of the result of a given expression, given partial information concerning the inputs. (For example, as discussed in Section 2.2.4, it is the *lack* of this information that frequently makes interval-based analysis highly inaccurate.) Because a symbolic representation can capture the entire expression for a value, the dependencies inherent *within* that expression can be explicitly represented. While this solves the problem in *theory*, in practice it does not: Maintaining full information concerning the dependencies of one value upon other values would require accumulating a complete record of the expressions and other constructs used to calculate a particular value. While this technique has been used in the literature for the analysis of small programs [Clarke 1981], it is hardly suitable for realistic analysis of even small code bases. To be feasible, analysis using symbolic predicates can only maintain a limited amount of other information regarding other values. This provides the representation with *partial* access to information regarding inter-value dependencies, but stops short of full information. Using this information, the system could (in principle) use predicates concerning

constituent values to calculate or bound the result of an expression. For example, given the information contained on the left of Figure 201, it is possible to more tightly bound the value for *c* than would otherwise be the case.

```
{ a>5, b ≤ 4 }, c = (b*b)/a ⇒ 0 ≤ c ≤ 3.2
```

Figure 201: Reasoning about Dependency-Based Information in the Symbolic Domain.

The second domain about which information is contained implicitly by a symbolic representation is the frequency with which different values could be expected to take place. Although the same constraints on representational capacity apply as in the case of information on dependencies, the symbolic predicates maintained concerning program values can in certain circumstances give insight into the relative frequency with which different possible values are observed during program execution.

2.4.3 Summary

The class of value abstractions that we have clumped together as "symbolic representations" is a diverse lot, and our commentary has of necessity been somewhat less precise and more tentative than in other discussions. Nonetheless, the discussion has revealed certain aspects of symbolic representations that are shared by all members of this class. In general, such representational make use of the full power of a symbolic *language* to describe constraints on values. This allows such values to be very expressive. Symbolic representations can in principle capture any variety of information available during analysis – including the dependency and frequency information that eludes every other representation examined in this appendix. As a result, the use of such values can allow for extremely powerful analyses, but can also lead to high space demands. Sophisticated rules can recognize patterns in the symbolic representations and extract additional information.

At the same time, the very expressiveness of the predicate language and the potential rules that manipulate it can pose serious problems for the amount of space occupied by such representations, for the time required for value combinations and for the convergence of the fixed point iteration within the analysis. Even such simple operations as taking the fixed point of two values can be complicated when applied on symbolically represented values, and an analysis that maintains symbolic values balance between the needs to offer greater precision and to reduce the cost of the representation. This is likely to be a challenge indeed.

A further, less obvious challenge is to devise a predicate language semantics and reasoning rules that are sufficiently rich to accurately model and exploit the very substantial idiosyncrasies of a discrete representation. As was discussed in Section 2.4.1, reasoning accurately about program values in the

presence of the subtleties imposed by the discrete representation is likely to require a large and sophisticated model of programming language semantics, and substantial care. The many "gotchas" that arise when reasoning about values as they are *actually* represented and manipulated might further prevent many steps of reasoning that would normally be deemed routine. Although a verdict would be entirely preliminary, it is possible that this would be a sufficiently serious phenomenon that it would negate many (or most) of the advantages of maintaining symbolic value representations.

It is possible that predicate-based values coupled with symbolic reasoning could provide a means of acquiring sufficiently rich information on program dependencies so as to make truly high-precision analysis possible, and to justify the additional cost of maintaining a representation of partially known values. Hints of this approach appear in the literature (e.g. [Nguyen 1989]), and our investigations have not been sufficiently deep to determine whether this is indeed a plausible route. Nonetheless, as discussed in Appendix 2, there are a number of challenges that must be overcome in order to employ symbolic representations effectively. In particular, the predicate language and reasoning machinery must be sufficiently expressive and clever to allow deduction of tight bounds on program values (bounds that emerge through "brute force" within the sample space representation of Chapter 11). At the same time, the richness of the predicate language and symbolic reasoning must be sufficiently restrained that operations will be efficient and (most importantly) so that the analysis will converge reasonably quickly.

2.5 Conclusion

This chapter has reviewed a spectrum of approaches for representing approximating domain information for scalar values. The approaches discussed exhibit widely varying tradeoffs in size, efficiency of combination, domain height (and thus convergence time), precision, and other metrics characterized in Table 38. While the 3-Level representation adheres to a "binary epistemology" in which values are classified as entirely known or entirely unknown, each of the other three approaches are crafted to represent rich information concerning *partially known* values. The fundamental motivation for modeling of such values is the fact that approximating values with greater precision may allow collecting domains to acquire a more accurate and complete picture of program behavior.

The interval domain stands out among the partially known value representations in that it has a *fixed* domain structure that limits the domain to a particular level of precision. Both of the other domains (the set representation and predicate-based techniques) allow for adjustment of domain precision based on the particular precision/performance tradeoff desired by the user. Unfortunately, both the interval representation and the scalable set-based representation suffer from a fundamental flaw that undermines their desirability: Both lose significant precision when modeling value combination by virtue of failing to

recognize the presence of inter-value *dependencies*. As was shown in Section 11.4.1.3 and Section 1.2, accurate characterization of the range of a value combination is impractical without knowledge of the dependencies that obtain between values. Predicate-based techniques offer an alternative representation that can allow for sidestepping this problem, but only with development of a very rich predicate language whose implementation may not be fully practical. Chapter 11 discusses two additional "heavy-weight" approximating value domain representations that attempt to ameliorate some of the shortcomings associated with extent representations; unfortunately, these advantages come only at the cost of significant decrease in the efficiency of value combination.

Appendix 3 Code for Sample Domains

Code for the Sample Domains can be found at the following URL:

ftp://hing.lcs.mit.edu/pub/hacrat/Thesis/SampleDomains.zip

Appendix 4 Code for Execution-Time Model

Code for Model of Execution in Chapter 7 can be found at the following address:

ftp://hing.lcs.mit.edu/pub/hacrat/Thesis/ModelCode.zip

Bibliography

Aho, A. V., R. Sethi, et al. (1986). *Compilers: Principles, Tools, and Techniques*. Reading, MA, Addison-Wesley

Ayers, A. E. (1993). Abstract analysis and optimization of Scheme. Cambridge, MA, Massachusetts Institute of Technology: 176. PhD Thesis.

Chase, D., M. Wegman, et al. (1990). Analysis of Pointers and Structures. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*: 296-310.

Chen, L.-L. (1994). Efficient computation of fixpoints that arise in abstract interpretation. Urbana-Champaign, Univ. of Illinois. Urbana-Champaign: 109. PhD.

Chen, L.-L. and W. L. Harrison (1992). Efficient computation of the fixpoints that arise in complex program analysis. Urbana-Champaign, IL, Univ. of Illinois. Urbana-Champaign: 34. Technical Report CSRD 1245.

Clarke, L. a. R., D (1981). Symbolic Evaluation Methods for Program Analysis. *Program Flow Analysis: Theory and Applications*. S. S. Muchnick and N. D. Jones. Englewood Cliffs, NJ, Prentice-Hall.: 418.

Cousot, P. and R. Cousot (1977). Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*. New York, NY, ACM Press: 238-252.

Drake, A. W. (1967). *Fundamentals of applied probability theory*. New York, McGraw-Hill

Eberlein, R. (1998). *Vensim*, Ventana, Inc.

Engler, D. R. (1996). *VCODE: A Retargetable, Extensible, Very Fast Dynamic Code Generation System.* SIGPLAN '96 Conference on Programming Language Design and Implementation, Philadelphia, PA, Association for Computing Machinery.

Forrester, J. (1968). *Principles of Systems*. Cambridge, The MIT Press

Futamura, Y. (1971). "Partial Evaluation of Computation Process -- An Approach a Compiler-Compiler." *Systems, Computers, Controls* 2(5): 45-50.

Grant, B., M. Mock, et al. (1997). DyC: An Expressive Annotation-Directed Dynamic Compiler for C. Seattle, WA, University of Washington. Technical Report UW-CSE-97-03-03.

Horwitz, S., P. Pfeiffer, et al. (1989). Dependence Analysis for Pointer Variables. *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*: 28-40.

Jones, N. D. and S. S. Muchnick (1981). Flow analysis and optimization of LISP-like structures. *Program Flow Analysis: Theory and Applications*. S. S. Muchnick and N. D. Jones. Englewood Cliffs, NJ, Prentice-Hall: 102-131.

Leone, M. and P. Lee (1996). *Optimizing ML with Run-Time Code Generation*. ACM SIGPLAN '96 Conference on Programming Language Design and Implementation, Association for Computing Machinery.

Nguyen, J. (1989). A Type Inference Algorithm for the Language L. Cambridge, MA, Massachusetts Institute of Technology: 88. Masters Thesis.

Nickel, K., Ed. (1986). Interval Mathematics 1985. Lecture Notes in Computer Science. New York, Springer-Verlag.

Noel, F., L. Hornof, et al. (1996). Automatic, Template-Based Run-Time Specialization: Implementation and Experimental Study. Rennes, France, IRISA: 23. Internal Publication 1065.

Osgood, N. D. (1993). PARTICLE--automatic program specialization system for imperative and low-level languages. Cambridge, Massachusetts Institute of Technology: 228. Masters Thesis.

Sussman, G. J. (1999). Personal communication.

Taylor, H. I. (1995). Personal communication.

Weise, D., R. Conybeare, et al. (1991). Automatic Online Partial Evaluation. *Functional Programming Languages and Computer Architecture (LNCS 523)*. J. Hughes. Cambridge, MA, Springer-Verlag, ACM: 165-191.

Weise, D. and E. Ruf (1990). Computing types during program specialization. Palo Alto, CA, Computer Systems Laboratory, Stanford University. Technical Report CSL-TR-90-441.

Yi, K. and W. L. Harrison (1993). *Automatic generation and management of interprocedural data flow analysis*. ACM 20th Symposium on Principles of Programming Languages, Charleston, South Carolina, Association for Computing Machinery.