A Relational Framework for Bounded Program Verification

by

Gregory D. Dennis

B.S., Massachusetts Institute of Technology (2002)
M.Eng., Massachusetts Institute of Technology (2003)

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 and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2009

Author .....
Department of Electrical Engineering and Computer Science
June 19, 2009

Certified by..
Daniel N. Jackson
Professor
Thesis Supervisor

Accepted by .....
Professor Terry P. Orlando
Chair, Department Committee on Graduate Students

A Relational Framework for Bounded Program Verification

by

Gregory D. Dennis

Submitted to the Department of Electrical Engineering and Computer Science
on June 19, 2009, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science and Engineering

Abstract

All software verification techniques, from theorem proving to testing, share the common goal of establishing a program's correctness with both (1) a high degree of confidence and (2) a low cost to the user, two criteria in tension with one another. Theorem proving offers the benefit of high confidence, but requires significant expertise and effort from the user. Testing, on the other hand, can be performed for little cost, but low-cost testing does not yield high confidence in a program's correctness.

Although many static analyses can quickly and with high confidence check a program's conformance to a specification, they achieve these goals by sacrificing the expressiveness of the specification. To date, static analyses have been largely limited to the detection of shallow properties that apply to a very large class of programs, such as absence of array-bound errors and conformance to API usage conventions. Few static analyses are capable of checking *strong* specifications, specifications whose satisfaction relies upon the program's precise behavior.

This thesis presents a new program-analysis framework that allows a procedure in an object-oriented language to be automatically checked, with high confidence, against a strong specification of its behavior. The framework is based on an intermediate relational representation of code and an analysis that examines all executions of a procedure up to a bound on the size of the heap and the number of loop unrollings. If a counterexample is detected within the bound, it is reported to the user as a trace of the procedure, though defects outside the bound will be missed.

Unlike testing, many static analyses are not equipped with coverage metrics to detect which program behaviors the analysis failed to exercise. Our framework, in contrast, includes such a metric. When no counterexamples are found, the metric can report how thoroughly the code was covered. This information can, in turn, help the user change the bound on the analysis or strengthen the specification to make subsequent analyses more comprehensive.

Thesis Supervisor: Daniel N. Jackson
Title: Professor

Acknowledgments

Many people deserve my thanks and gratitude for making this thesis possible.

First, a big thanks to my supervisor Daniel Jackson. It was Daniel's teaching of 6.170 that ignited my interested in software design and development. After that class, I joined his Software Design Group as an undergraduate researcher, then a Masters student, and finally a Ph.D. Throughout that time, he has provided invaluable insights, challenges my assumptions, and made me think more deeply and clearly about every problem I encountered. Thanks, too, to the other members of my thesis committee: Carroll Morgan and Arvind. With their insights, edits, and tough questions, I was able to hone my thinking and clarify my presentation.

During my time in SDG, I have had the pleasure of meeting and working with many smart people. First, a thanks to the doctoral students who welcomed me into the group when I first joined and provided mentorship and advice along the way: Sarfraz Khurshid, Ilya Shlyakhter, and Mandana Vaziri. I think, too, all the fellow students and researchers who joined about the same time I did and whom I relied upon for advice and laughs: Felix Chang, Jonathan Edwards, Carlos Pacheco, Derek Rayside, Robert Seater, Mana Taghdiri, and Emina Torlak. Finally, thanks to all the new additions to the group, whose energy and enthusiasm has been a source of inspiration: Zev Benjamin, Eunsuk Kang, Aleksandar Milicevic, Joe Near, Rishabh Singh, and Kuat Yessenov.

Kuat deserves special thanks for his intense involvement building a Java front-end to my analysis. Without his tireless work understanding the Java Modelling Language and translating it to relational logic, the case studies would not have been possible. His more recent work on the JForge Eclipse plugin and specification language brought the usability and applicability of my research to an entirely new level. He has a bright future ahead.

Finally, I thank my family. My parents, brothers, and sister were always there for me with encouragement and support. My mom's warmth and my dad's judgement are the backbones of my success. Most of all, I thank my loving wife Joselyn, who during my time as a Doctoral student, married me, bought a home with me, and gave birth to our beautiful son, Noah. She has always been patient and supportive, and Noah will soon realize how lucky he is to have a mother like her.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software failures continue to pose a major problem to software development and the economy generally. Studies have estimated software errors to account for between 25% and 35% of system downtime [76, 71]. A 2002 study by the National Institute of Standards & Technology (NIST) estimated the annual cost of software failures in the U.S. at between $20 and $60 billion [68], or as much as 0.6% of the U.S. GDP. The consequences of these failures vary greatly, from the merely annoying, such as the crash of a desktop word processor, to the tragic, such as the 28 overdoses delivered by a radiotherapy machine in Panama City in 2001, an event which led to 17 deaths [44].

Software verification techniques hold the promise of dramatically reducing the number of defects in software. The term "software verification" refers here to all manner of checking program correctness: from formal verification approaches like theorem proving, to dynamic analyses like testing, to static analyses, such as ESC/Java. Ideally, these techniques would establish that a program is correct with a high degree of confidence while requiring limited cost from the user, but these two goals — high confidence and low cost — are at odds. Theorem proving, for example, offers the benefit of high confidence, but requires significant expertise and effort from the user. Testing, on the other hand, can be performed for little cost, but low-cost testing does not yield high confidence in a program's correctness.

1.1 The Cost vs Confidence Tradeoff

Our assessment of the cost-versus-confidence tradeoff of testing and theorem proving is shown in Figure 1-1. "Cost" in the figure refers to both the time required to carry out the technique and the cost of employing someone with the expertise to carry it out. "Confidence" refers to the degree of certainty in the program's correctness that the analysis provides.

Due to its low initial costs, testing can be an attractive option for checking program correctness. A handful of test cases can be written quickly without much expertise, run fully automatically, and when bugs are found, testing provides the user with concrete counterexample traces that exhibit the error. As shown in Figure 1-1, testing allows users, for relatively little cost, to obtain an immediate boost of confidence in the correctness of their code.

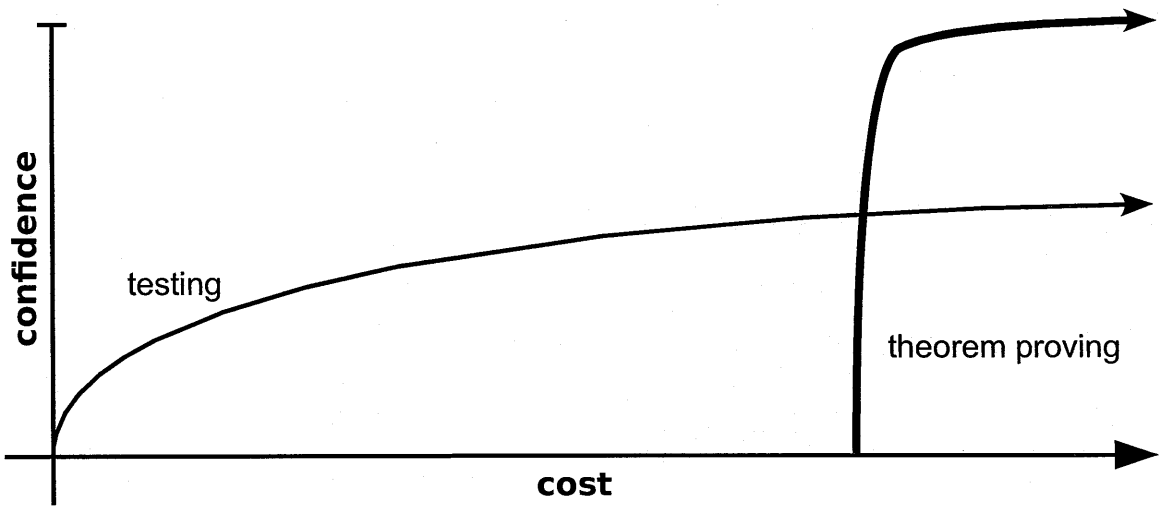


Figure 1-1: Cost vs. Confidence comparison of testing and theorem proving.

The challenge of testing is that high confidence is very difficult and costly to obtain. Test selection, test generation, and calculation of coverage metrics are tasks too tedious to be performed manually, and building testing infrastructure to automate them causes dramatic increases in cost. As a result, the testing curve in Figure 1-1 quickly levels off. The eventual high costs of testing explain why it consumes about half the total cost of all software development [11] and why Microsoft, for example, employs approximately one tester for every developer [17].

If very high confidence is required, a developer might instead opt for theorem proving. As its name implies, theorem proving yields a proof of the program's correctness — a very high level of confidence. However, as reflected in Figure 1-1, theorem proving requires a large investment in both time and expertise before any confidence is obtained. It is common for a theorem prover to carry a learning curve of about 6 months, even for a highly skilled software developer [87, 6]. That amount of upfront investment makes theorem proving prohibitively expensive for most projects.

1.2 Strong vs Weak Specifications

One area with much recent progress in combining high confidence with low cost is static analysis. A variety of static-analysis tools can now automatically check a program for conformance to a specification and provide a high degree of confidence in their result. Some of these tools, like FindBugs [2], use pattern-based heuristics; others, such as Astrée [1], use abstract interpretation; some combine abstract interpretation and model checking, like SLAM [8]; and others are explicit-state model checkers, like SPIN [43].

However, these analyses, by and large, achieve their combination of full automation and high confidence by sacrificing the expressiveness of the specifications they accept. To date, static analyses have largely been limited to checking code against very partial properties that apply to a large class of programs, such as the absence of array-bound errors and conformance to API usage conventions. Some shape analyses, like SATC [42], can check deeper, structural properties of the heap, such as list acyclicity, but these are nevertheless still very partial correctness properties.

Less progress has been made in static analyses for checking *strong specifications*. A "strong specification" here refers to a detailed property that is specific to the individual program under analysis.[1] For the software controlling a radiotherapy machine, a strong specification may be that "the radiotherapy machine delivers the prescribed dose," and for a voting system perhaps "the winner of the election has the most votes." These are properties that require the precise behavior of the code for their satisfaction, and thus cannot be analyzed by a technique that relies on abstraction, a technique common to static analysis generally. Strong specifications are typically written by the user in a general-purpose specification language, such as Larch [41], the Java Modeling Language [58], or Spec# [10].

To summarize, while many static analyses deliver high confidence at a low cost, they typically do so with respect to a partial, general-purpose property, not a strong, program-specific specification. As a result, the high confidence they offer in the program's correctness with respect to that property does not translate into high confidence in its correctness overall.

1.3 Introducing Forge

While there is much research dedicated to shifting the testing curve in Figure 1-1 up (to increase its confidence) and other research to shift the theorem proving curve to the left (to lower its cost), the analysis presented in the this thesis, implemented in a tool called *Forge*, aims for a different cost/confidence tradeoff. As illustrated in Figure 1-2, it requires a greater upfront investment than testing, but once that initial investment is made, it provides a dramatic increase in confidence. Being fully automatic, its cost is much lower than theorem proving, though it can never attain the confidence of a proof. And unlike most automatic static analyses, which sacrifice the expressiveness of the specifications they accept, Forge is capable of checking a procedure against a strong, user-provided property of its behavior.

We expect Forge to be economical for those software projects whose current investment falls somewhere in the gray area in Figure 1-2 but are currently situated on the testing curve. The black circle in the figure is example of where those projects might currently lie, and the arrows point to where they might move in the space if they were to adopt Forge. For their current level of investment, Forge could significantly increase the level of confidence in the software's correctness. Or, if the current level of confidence is already deemed adequate, Forge could obtain that confidence for lower cost.

[1]The term "strong specification" is borrowed from Robby, Rodríguez, Dwyer, and Hatcliff [72]

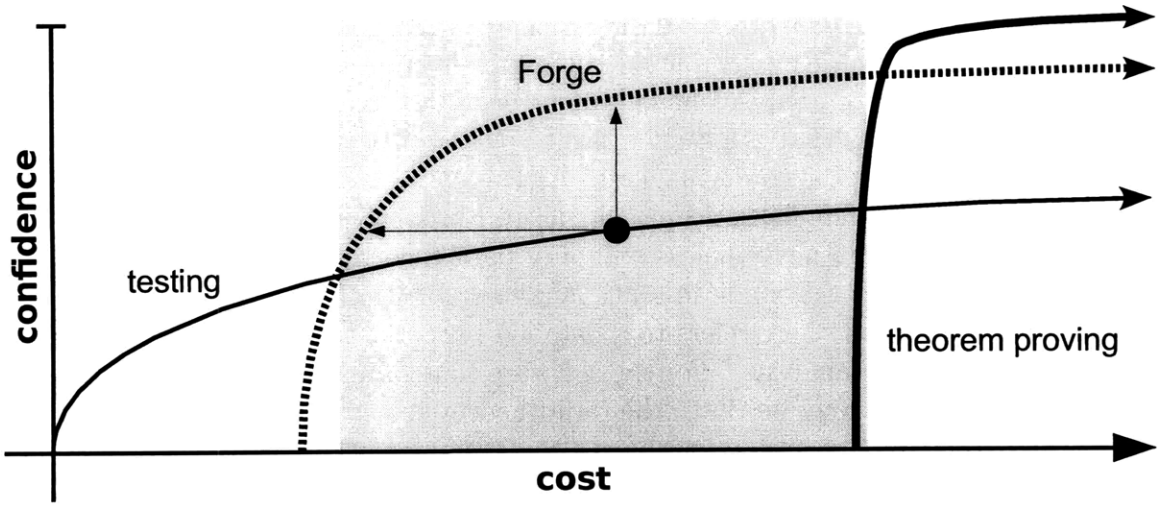


Figure 1-2: The Forge Tradeoff. Forge is economical for those development projects spending anywhere in the gray area on testing. For that cost, Forge offers greater confidence, and for that level of confidence, Forge could offer a lower cost.

1.3.1 The Forge Framework

Forge is a program-analysis framework that allows a procedure in a conventional object-oriented language to be checked against a strong specification of its behavior. The Forge framework is shown in Figure 1-3. The elements in black in the framework are the contributions of this thesis; elements in the gray are the contributions of others; and those in white have yet to be developed.[2]

At the center of the framework sits the Forge tool. Forge accepts as input a procedure, a specification of the procedure, and a bound on the analysis. The procedure and its specification are both expressed in the Forge Intermediate Representation, or FIR. Discussed in greater detail in Chapter 2, FIR is a simple relational programming language that is amenable to analysis. FIR is intended to be produced by a mechanical translation from a conventional high-level programming language, but a user could also program in FIR directly. Specifications in FIR can involve arbitrary first-order logic, plus a relational calculus that includes transitive closure.

The bound provided to Forge consists of the following:

- a number of times to unroll each loop and recursive call;
- a bitwidth limiting the range of FIR integers; and
- a *scope* for each type, i.e., a limit on the number of its instances that may exist in any heap reached during execution.

[2]JForge was a collaborative effort with Kuat Yessenov.

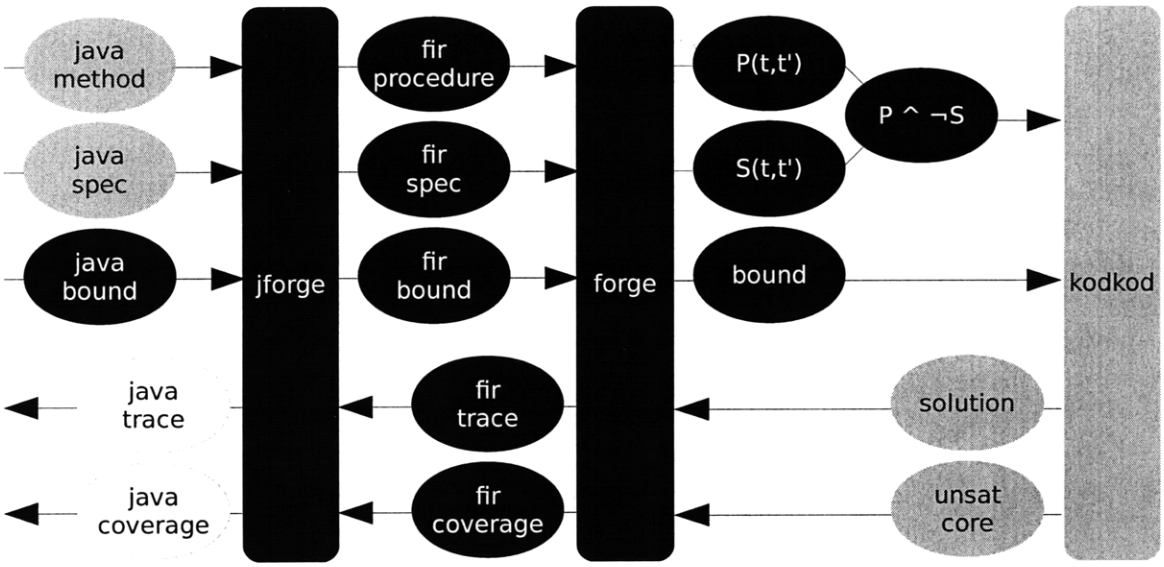


Figure 1-3: The Forge Framework. Elements in black are the contributions of this thesis; gray are the contributions of others; and white have yet to be developed.

Forge searches exhaustively within this bound for a trace of the procedure that violates the specification. Bugs outside the bound will be missed, but in exchange for that compromise, the analysis is fully automated and delivers concrete counterexamples. The success of this *bounded verification* approach rests on an idea called the "small-scope hypothesis," which is the expectation that bugs will usually have small examples. It's a hypothesis consistent with our own experience, with prior empirical evaluation [5, 27, 28], and with the case studies presented in Chapter 6 of this thesis.

To perform the exhaustive search, Forge encodes the FIR procedure and specification in the relational logic accepted by the Kodkod model finder [84]. From the procedure, Forge automatically obtains a formula $P(s, s')$ in this logic that constrains the relationship between a pre-state s and a post-state s' that holds whenever an execution exists from s that terminates in s'. The states are vectors of free relational variables declared in the logic. The translation of the procedure to the logic uses a symbolic execution technique presented in Chapter 3. A second formula $\psi(s, s')$ is obtained from the provided specification, the negation of which is conjoined to P yielding the following formula, which is true exactly for those executions that are possible but that violate the specification.

$$P(s, s') \wedge \neg\psi(s, s') \tag{1.1}$$

This formula is equivalent to the negation of $(P(s, s') \Rightarrow \psi(s, s'))$, the claim that every execution of the procedure satisfies the specification. Forge translates the bound on the FIR procedure into a bound on the relations in states s and s' and hands the formula and resulting bound to Kodkod. Using a SAT solver, Kodkod exhaustively searches for a solution to the formula within the bound, and reports such a solution if one exists. Forge then translates this solution back into a counterexample trace of the FIR procedure.

Since bounded verification is unsound, when no counterexamples are found it is unclear how much confidence one should have that the procedure meets its specification. For example, was the bound on the analysis too small to fully exercise all of the procedure's behaviors? Testing addresses this problem with coverage metrics, analyses which can reveal behaviors of code that a test suite has left unexplored, but most static analyses offer nothing analogous.

Forge includes its own coverage metric, analogous to those for testing, which, in the absence of counterexamples, can identify a subset of the code that was not explored. This information can, in turn, inform the user how to tune Forge to attain a more comprehensive analysis. The coverage metric, which is based on an "unsatisfiable core" formula reported by Kodkod, is explained in detail in Chapter 4.

Since Forge operates on code and specifications written in the intermediate representation, analyzing a conventional high-level language requires a front-end for translating that language and its specifications to FIR. The Forge framework includes *JForge*, a front-end tool for translating Java code and specifications written in the JForge Specification Language [90] into FIR. A front-end for C is being developed independently by Toshiba Research [75]. Chapter 5 explains how a high-level object-oriented programming language like Java is translated to FIR. As will be explained, there are some limitations in this translation: it cannot handle some features of Java, including multithreading and real numbers, and it uses a few unsound optimizations that can theoretically lead to false alarms (Section 5.1.3). It is left to future work on JForge to map the trace and coverage information produced by Forge back to the original Java code.

When analyzing object-oriented programs, an additional complexity arises from the specification of abstract data types. Note that in Formula 1.1, the procedure and specification are formulas over the same state, but when checking a procedure of an abstract data type, the (abstract) representation referred to in the specification usually differs from the (concrete) representation in the code. Chapter 5 addresses how to make such specifications amenable to the bounded verification analysis.

We have conducted a series of case studies with Forge demonstrating its utility and feasibility, and these are presented in Chapter 6. Finally, related work, directions for future work, and general conclusions are discussed in Chapter 7. But before embarking on these technical presentations, the following section demonstrate how the Forge framework looks from the user's perspective.

1.4 Forge from the User's Perspective

This section illustrates a user's experience with the Forge framework on a small example. The example involves checking a method of a linked-list implementation against its specification. The code and specification of the linked list is shown in Figure 1-4. The linked list is circular and doubly-linked, and for simplicity, the buckets in the list are themselves the values[3]. Clients of the list are expected to extend the abstract `Value` class with the values they wish to store in the list.

[3]The implementation is similar to the `TLinkedList` class provided by the GNU Trove library [3].

In this example, our hypothetical user wishes to check the `get` method of the linked list against its specification. A bug has deliberately been seeded in the `get` method: in the body of the first for-loop, `value` should be assigned `value.next` instead of `value.prev`. Because this bug does not raise a runtime error, tools that search for shallow properties like the absence of null pointer dereferences and array out-of-bounds errors cannot catch it. To detect the bug, one must apply an analysis that is capable of checking a code against a strong specification of its behavior.

The strong specifications found in Figure 1-4 have been written by our user in the JForge Specification Language (JFSL)[4] [90]. JFSL is not a contribution of this thesis and so will not be discussed in depth, but enough will be explained to understand this example. JFSL specifications are written in the form of Java *annotations*, which are special Java metadata constructs that begin with the '`@`' symbol.

The `SpecField` annotation at the top of Figure 1-4 declares a *specification field* named `elems` that serves as the abstract representation of the list data type. The `elems` field is of type `seq Value`, meaning it is a sequence of `Value` objects. In JFSL, a sequence declared of type `seq T` is actually a binary relation mapping integers to elements drawn from type `T`, whose domain is constrained to consist of consecutive integers, starting at zero. So `elems`, while conceptually a sequence, is semantically a binary relation of type Integer $\rightarrow$ Value. For example, the sequence `[A, B, C]` would be encoded in `elems` as the binary relation $\{\langle 0,A\rangle, \langle 1,B\rangle, \langle 2,C\rangle\}$.

The formula given in a `SpecField` annotation is the abstraction function that determines the value of the specification field from the values of the concrete fields. The abstraction function in Figure 1-4 gives an inductive definition of `elems` as a conjunction of two constraints. According to the first constraint (the base case of the induction), the first element in the sequence equals the set-difference of the `head` field and `null`. When `head` is null, there is no first element (`this.elems[0]` is the empty set); and when `head` is not null, the first element equals `head`. The second constraint (the inductive case) says that for every positive integer `i`, the element at index `i` in the sequence equals the `next` field of the element at index `(i - 1)`, unless `next` equals `head`, which indicates the end of the list has been reached.

The representation invariant of `LinkList` is given by the `Invariant` annotation, where it is expressed as two (implicitly conjoined) constraints. According to the first constraint, when the `head` field is not null, the `head` is reachable from itself by following the `next` field one or more times. The expression `^next` denotes the transitive closure of the `next` field. The second constraint says the `size` field is equal to the number of reachable values in the list. The '`#`' symbol is the cardinality operator: `#e` yields the number of tuples in the relation `e`.

The specification of the `get` method is expressed in three annotations on the method: `Requires`, `Returns`, and `Throws`. The `Requires` annotation specifies a precondition under which no exception is thrown, in this case it is that the `index` argument is within the bounds of the list, i.e., it is non-negative and less than the length of the list. The expression `#this.elems` yields the number of tuples in the sequence, which is the same as the length of the sequence. The `Returns` annotation

[4] JFSL is a product of ongoing work led by Kuat Yessenov.

```
@SpecField("elems: seq Value | " +
        "(this.elems[0] = this.head - null) && " +
        "(all i: int | i > 0 => this.elems[i] = this.elems[i-1].next - this.head)")
public class LinkList {

    @Invariant({"this.head != null => this.head in this.head.^next",
                "#this.head.^next = this.size"})
    private Value head;
    private int size;

    @Requires("index >= 0 && index < #this.elems")
    @Returns("this.elems[index]")
    @Throws("IndexOutOfBoundsException: index < 0 || index >= #this.elems")
    public Value get(int index) {
        // check whether index is in bounds
        checkIndex(index);

        Value value;
        // if index is in front half of list,
        // search from the first element
        if (index < (size >> 1)) {
            value = head;
            for (int i = 0; i < index; i++) {
                value = value.prev;
            }
        }
        // if index is in back half of list,
        // search from the second-to-last value
        else {
            value = head.prev;
            for (int i = size - 1; i > index; i--) {
                value = value.prev;
            }
        }
        return value;
    }

    @Requires("index >= 0 && index < #this.elems")
    @Throws("IndexOutOfBoundsException: index < 0 || index >= #this.elems")
    private void checkIndex(int index) {
        if (index < 0 || index >= size) {
            throw new IndexOutOfBoundsException();
        }
    }

    @Invariant("next = ~prev")
    public static abstract class Value {
        Value next, prev;
    }
}
```

Figure 1-4: Linked List Implementation & Specification

specifies an expression to be returned by the method when the precondition given by `Requires` is true. The expression in the `Returns` annotation, `this.elems[index]`, evaluates to the `Value` at the given `index` in the `elems` sequence. The `Throws` annotation specifies that the method throw an `IndexOutOfBoundsException` when `index` is out of bounds.

The specification of `checkIndex` has the same `Requires` and `Throws` annotations as the specification of `get`, because it throws the same exception under the same conditions. Lastly, the inner class `Value` has an invariant that the `next` field is the relational transpose ($\sim$) of the `prev` field. In other words, if following the `next` field of `x` yields `y`, then following the `prev` field of `y` must yield `x`. Code that is not relevant to the analysis of the `get` method is not shown in Figure 1-4.

1.4.1 An Example Analysis

To perform an analysis with Forge, the user provides a bound on the analysis, consisting of a number of loop unrollings, a bitwidth to restrict the range of integers, and a *scope* on each type. The "scope" of a type is a limit on the number of instances of that type that may exist over the course of an execution. For this example, our hypothetical user applies JForge to analyze the `get` method initially in a scope of 3 `Value` instances and 1 `LinkList`, a bitwidth of 4, and 1 loop unrolling.

The analysis completes in about two seconds and reports that "no counterexamples were found." Not sufficiently confident that no bugs exist, our user invokes Forge's coverage metric, which responds by highlighting statements that were *missed* (not covered) by the prior analysis. A statement is "missed" if the analysis would have still succeeded had been removed from the procedure. For our example analysis, the coverage metric highlights the first for-loop as missed:[5]:

```
. . .
if (index < (size >> 1)) {
  value = head;
  for (int i = 0; i < index; i++) {
    value = value.prev;
  }
}
. . .
```

From this coverage information, our user realizes that her chosen bound of 3 `Value` instances was too small for adequate coverage, because the first for-loop is unnecessary for lists of length 3 or less. If the list is of length 3, then index arguments equal to 1 or 2 are reached from traversing from the end of the list. Only an index of 0 is reached by iterating from the front-half of the list, and when the for-loop is reached with an index of 0, the loop condition is immediately false, rendering the entire loop unnecessary.

[5]The metric highlights statements in FIR code but is not yet mapped to Java source code.

```
public Value get(int index) {
    checkIndex(index);
    Value value;
    if (index < (size >> 1)) {
        value = head;
        for (int i = 0; i < index; i++) {
            value = value.prev;
        }
    }
    else {
        value = head.prev;
        for (int i = size - 1; i > index; i--) {
            value = value.prev;
        }
    }
    return value;
}
```

```
initial state:
    LinkList = {L0}
    Value = {V0, V1, V2, V3}
    this = L0
    index = 1
    elems = {<L0, 0, V0>, <L0, 1, V1>, <L0, 2, V2>, <L0, 3, V3>}
    head = {<L0, V0>}
    next = {<V0, V1>, <V1, V2>, <V2, V3>, <V3, V0>}
    prev = {<V1, V0>, <V2, V1>, <V3, V2>, <V0, V3>}
    size = {<L0, 4>}
checkIndex(index):
if (index < (size >> 1)):
    true
value = head:
    value = V0
int i = 0:
    i = 0
i < index:
    true
value = value.prev:
    value = V3
i++:
    i = 1
i < index:
    false
return value:
    return = V3
```

Figure 1-5: The get method and its counterexample trace.

Desiring a more thorough analysis, our user increases the scope of `Value` to four instances and runs Forge again. The analysis again completes in two seconds, but this time finds a trace of the `get` method that violates its specification. This trace is shown in Figure 1-5. This trace is an execution of the `get` method where the `index` argument is 1 and the list is the sequence `[V0, V1, V2, V3]`. The return value should therefore be `V1`, but it is incorrectly `V3`.

By inspecting the trace, the user discovers the error (`value` is assigned `value.prev` in the first loop instead of `value.next`). After fixing the error, the user repeats the Forge analysis and it reports that counterexamples are no longer found. However, the tool still shows some statements missed by the analysis. The user increases the scope to 5 and then finally to 6 `Value` instances, at which point the analysis completes in two minutes and reports full coverage. (The analysis time when coverage mode is turned off is 10 seconds.)

Although the analysis has reached full coverage and no counterexamples are found, it has not established a proof of correctness. Full coverage indicates only that the coverage analysis has reached the limit of its ability to discover areas where bugs may exist.

1.5 Discussion

The linked list example makes concrete several of the features and characteristics of the Forge framework. First and foremost, it demonstrates the ability of Forge to check a method in an object-oriented program against a strong specification of its behavior. Even many tools that ostensibly check strong specifications, such as ESC/Java2 [23], would not be able to handle the specification in this example, because it used transitive closure, a feature which they do not support.

The example showcased some advantages of Forge over theorem provers. Once the bound was chosen, the analysis was fully automatic and did not require user interaction or the writing of loop invariants as required by verification techniques, like Boogie [9] for example. Also, the code failures are presented as counterexample traces, rather than failed verification conditions or open subgoals that theorem provers often report. The traces make locating the error a relatively easy task.

A test suite that achieves full branch coverage would have probably found the bug, but not necessarily. Indeed, if the sequence has duplicates, then with the right argument, the method could return the correct value even when a test case branches into the first for-loop to execute the erroneous statement. Although that is unlikely, the observation highlights the fact that a test suite that achieves full branch or path coverage makes no claim about which heap configurations have been explored. Bounded verification, in contrast, makes a specific claim about the heaps explored — all those within the user-provided bound — and when combined with the coverage metric illustrated, makes a claim about code coverage as well.

The initial analysis in a scope of three value instances and a bitwidth of 4 explored all lists up to length 7 with 3 unique elements. There are $\sum_{k=0}^{7} 3^k = 3280$ such lists and and 16 different integer arguments, for a total of $16 \times 3280 = 52,480$ argument combinations explored, a large number of test cases to write and execute. With Forge, all those tests were effectively constructed and simulated in two seconds. The final analysis in a scope of 6 values explored the equivalent of $16 \times \sum_{k=0}^{7} 6^k = 5,374,768$ tests and the analysis completes in 10 seconds. Granted, many of those lists are isomorphic to one another; by our calculation there are 1155 non-isomorphic lists up to length 7 for a total of 18480 test cases. But avoiding non-isomorphic tests in general would pose an additional burden on a tester. Forge is able to avoid searching through many isomorphic structures for free by relying on the symmetry breaking capabilities of the underlying Kodkod model finder.

Forge's bounded verification analysis is unsound: when it does not find a trace of the procedure that violates the specification, that is not a guarantee of the program's correctness. However, the coverage analysis helps mitigate that unsoundness. In our example, coverage reported by Forge showed that the initial analysis had not exercised the statement inside the first for-loop, which motivated our hypothetical user to expand the bound, thereby leading to the bug's detection.

1.6 The Road Ahead

The remainder of the thesis describes the techniques that make this user experience possible. It covers the following topics:

- the intermediate representation on which the Forge analyses are performed;
- the bounded verification analysis that given a procedure and specification in the intermediate representation searches for a trace of the procedure that violates the specification;
- the coverage metric that reports how thoroughly the bounded verification analysis examined the code;
- techniques for applying the Forge analysis to programs written in high-level languages;
- case studies in which Forge was used to analyze Java programs; and
- discussion of related work, suggestions for future directions, and general reflections on the tool.

Enjoy!

Chapter 2

Intermediate Representation

Due to the complexities of dealing with high-level programming languages, many program verification techniques encode high-level programs in an intermediate representation (IR) that is more amenable to analysis [59]. ESC/Java [37] and ESC/Java2 [23] encode Java in a variant of Dijkstra's guarded commands [31]; Boogie [9] encodes .NET bytecode in BoogiePL [25]; the Bogor model checker [72] encodes Java in the Bandera Intermediate Representation [46]; and Krakatoa [62] and Caduceus [36] encode Java and C, respectively, into the Why language [35]. These intermediate representations facilitate transformations and optimizations of the code, and they simplify the eventual translation to verification conditions.

The Forge Intermediate Representation (FIR) is the language on which the Forge bounded verification and coverage analyses are performed. In contrast to other intermediate representations, FIR is *relational*. That is, every expression in the language evaluates to a relation, a feature that makes its semantics simple and uniform, and therefore, more amenable to automatic analysis. In addition to being a programming language, FIR is at the same time a *specification* language. As will be illustrated in this chapter, declarative specification can be embedded as statements — *specification statements* — within what is otherwise imperative code. And FIR expressions can include arbitrary first-order logic (any alternation of quantifiers), a useful, if not necessary, feature for writing strong specifications.

Modeling code and specifications with a combination of first-order logic and a relational calculus is an idea drawn from experience with the Alloy modeling language [48] and the Kodkod model finder [84]. User experience with Alloy has shown that a combination of first-order and relational logic can encode the heap of an object program and operations on that heap in a clear and concise way [47]. Further experience and empirical data [82] has demonstrated the Kodkod model finder to be an efficient tool for finding solutions to formulas in this logic.

To analyze programs written in a conventional high-level programming language they must first be translated to FIR. The Forge framework includes a translation from Java to FIR that is discussed in Chapter 5, and Toshiba Research is developing a translation for C [75]. This chapter describes the structure and semantics of FIR, as well as and how to unroll loops and inline method calls in FIR procedures.

Program ::= UserDomain* UserLiteral* Variable* Procedure*

Domain ::= Boolean | Integer | UserDomain
Type := ∅ | Domain | Type ∪ Type | Type → Type
UserDomain ::= **domain** domId

Variable ::= LocalVariable | GlobalVariable
LocalVariable ::= **local** localId: Type
GlobalVariable ::= **global** globalId: Type
varId ::= localId | globalId

Literal ::= BooleanLiteral | IntegerLiteral | UserLiteral
BooleanLiteral ::= true | false
IntegerLiteral ::= 0 | 1 | -1 | 2 | -2 | ...
UserLiteral ::= **literal** litId: domId

Procedure ::= **proc** procId (localId*) : (localId*) Stmt
Stmt ::= BranchStmt | UpdateStmt;Stmt | ExitStmt
BranchStmt ::= **if** Expr **then** Stmt **else** Stmt
UpdateStmt ::= Assign | Create | Call | Spec

Assign ::= varId := Expr
Create ::= varId := **new** domId
Call ::= varId* := procId (Expr*)
Spec ::= varId* := **spec** (Expr)

Expr ::=	
varId | litId | domID | ∅	leaf
| Expr ⊆ Expr	subset
| Expr {= | ≠} Expr	(in)equality
| {**some** | **one** | **lone** | **no**} Expr	multiplicity
| Expr {∪ | ∩ | \} Expr	set operations
| Expr . Expr	join
| Expr → Expr	cross product
| Expr ⊕ Expr	override
| ^Expr	transitive closure
| ~ Expr	transpose
| π(Expr, IntegerLiteral*)	projection
| Expr ? Expr : Expr	conditional
| {varId* | Expr}	comprehension
| ¬ Expr	Boolean negation
| Expr {∧ | ∨} Expr	Boolean operations
| {∀ | ∃} varId* | Expr	quantification
| Σ varId* | Expr	summation
| Expr {+ | − | × | ÷ | mod} Expr	arithmetic
| Expr {> | < | ≥ | ≤} Expr	integer inequality
| Expr {| | & |^| ≪ | ≫ | ⋙ } Expr	bitwise operations
| # Expr	cardinality
| $varId_{old}$	pre-state variable

Figure 2-1: FIR Grammar

2.1 The FIR Grammar

The FIR grammar is shown in Figure 2.1. The Forge Intermediate Representation is described as "relational", because every expression in its grammar evaluates to a *relation*. A relation is a set of *tuples*, where each tuple is a sequence of *atoms*. The arity of a relation (the length of its tuples) can be any strictly positive integer. A set of atoms can be represented by a unary relation (relation of arity 1), and a scalar by a singleton set.

FIR consists of data structures assembled via API calls and has no formal syntax. However, for expository purposes, this thesis includes textual and graphical representations of these data structures as needed. Figure 2.1 shows a textual representation of a FIR program that performs registration for a website on which every user must have a unique email and a unique integer id. The register procedure takes an email argument and returns a user atom. If an existing user already has that email, the procedure returns the Error literal. Otherwise, the procedure creates and returns a new user instance with that email and with a new unique id.

As shown in Figure 2.1, a *program* declares a series of *user-defined domains*, *user-defined literals*, *variables*, and *procedures*. A *domain* is a sort ("sort" as in "sorted logic"), a set of atoms that is disjoint from all other domains. Two domains are built into the language: the domain of Boolean values and the domain of integers[1]. A FIR program may declare any number of *user-defined domains*, which are the only domains from which new atoms may be dynamically allocated. The example in Figure 2-2 declares two user-defined domains, User and String.

[1]FIR does not currently provide a domain of real numbers, though support for real numbers is a potential area of future research.

```
domain User, domain String, literal Error: User
global id: User→Integer, global email: User→String
local newEmail: String, local newUser: User, local newId: Integer

proc register (newEmail) : (newUser)
1  if newEmail ⊆ User.email
2     newUser := Error
   else
3     newUser := new User
4     email := email ∪ newUser→newEmail
5     setUniqueId(newUser)
6  exit

proc setUniqueId (newUser) : ()
7  id := spec(∃ newId | (id = id_old ⊕ newUser→newId) ∧ ¬(newId ⊆ User.id_old))
8  exit
```

Figure 2-2: Website Registration Program in FIR

The *type* of a FIR expression is either a domain or some combination obtained by unions and cross products of domains. For example, an expression of type $(D_1 \cup D_2) \rightarrow D_3$ evaluates to a binary relation, whose first column contains atoms from domains D_1 and D_2 and whose second column is drawn from domain D_3. An expression with the empty type ($\emptyset$) must evaluate to the empty set[2].

FIR variables, both global and local, are declared with an identifier and a type. The program above declares two global variables: id and email. The id global variable is declared of type User → Integer, meaning it is a binary relation mapping users to integers. Similarly, email of type User → String is a binary relation from users to strings. (If this FIR program has been generated from high-level object-oriented code, id and email likely correspond to fields named `id` and `email` in a class named `User`.)

A FIR program declares a single alphabet of local variables to be used by the procedures. Semantically, every procedure gets its own copy of every local variable. For example, the register and setUniqueId procedures both use the newUser and newId local variables, but they are using their own copy of those variables, not accessing shared state as they would if these variables were global. All the local variables in the example are declared to be sets (relations of arity 1) but there is no restriction in FIR that this be the case. Local variables may in general be relations of any arity, just like global variables (although multiple-arity local variables never appear in FIR that is generated by JForge).

[2]Therefore, an expression with the empty type, unless that expression is the empty set itself, indicates a likely error in the generation of the FIR code.

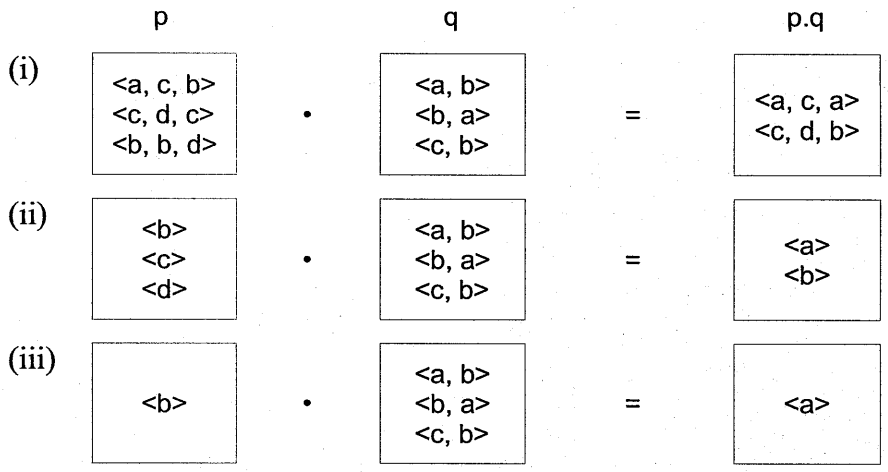


Figure 2-3: Three Examples of Relational Join. For each tuple in p of the form $\langle p_1, \ldots, p_n, m \rangle$ and tuple in q of the form $\langle m, q_1, \ldots, q_k \rangle$, there is a tuple in p.q of the form $\langle p_1, \ldots, p_n, q_1, \ldots, q_k \rangle$. Example (ii): when p is a set, the join p.q is equivalent to the relational image of q under p. Example (iii): when p is a singleton and q is a function, p.q is equivalent to function application.

Literals are constant values that evaluate to singleton sets, and the value of each literal is disjoint from every other literal. FIR has built-in literals for the Boolean constants true and false and for each integer. Programs may additionally declare *user-defined literals* belonging to any of the declared user-defined domains. The FIR program above declares a literal named Error of type User, a value that the register procedure returns to signal an error has occurred.

2.1.1 FIR expressions

To simplify automatic analysis, every expression in FIR is side-effect free. Domains, variables, literals are all expressions themselves. When treated as an expression, a domain evaluates to its extent, i.e., the set of atoms that have so far been allocated from the domain. The empty type ($\emptyset$) doubles as the expression for the empty set. The expression language has largely been adopted from the Alloy Modelling Language [48].

An important operator in the expression grammar is the relational join (.). Informally, the join expression p.q matches the right-most column of p against the left-most column of q, concatenates the matching tuples, and drops the matching column. Formally: for each tuple in p of the form $\langle p_1, \ldots, p_n, m \rangle$ and tuple in q of the form $\langle m, q_1, \ldots, q_k \rangle$, there is a tuple in p.q of the form $\langle p_1, \ldots, p_n, q_1, \ldots, q_k \rangle$.

Three examples of applying the relational join are shown above in Figure 2-3. In example (i), a ternary relation p is joined with a binary relation q. Example (ii) shows that when p is a set, the join p.q is equivalent to the relational image of q under p. For example, the register procedure contains the expression User.email, an expression which evaluates to the set of all emails currently held by some user. Example (iii) shows that when p is a singleton and q is a function, p.q is equivalent to function

application. The expression newUser.email, for example, evaluates to the singleton set containing the email of newUser.

Expressions can involve standard set operations, too, including union ($\cup$), intersection ($\cap$), difference ($\setminus$), and cross product ($\rightarrow$). For example, Statement 3 in the register contains the expression

$$\mathsf{email} \cup \mathsf{newUser} \rightarrow \mathsf{newEmail}$$

Suppose that newUser is `Greg`, that newEmail is `gdennis@mit.edu`, and that email currently contains no tuple beginning with `Greg`. The cross product newUser$\rightarrow$newEmail, evaluates to a relation containing the single tuple <`Greg`, `gdennis@mit.edu`>, so the entire expression evaluates to a relation that is the same as email except it also contains that tuple.

Another frequently used operator is relational override ($\oplus$). The expression $\mathsf{p} \oplus \mathsf{q}$ denotes all the tuples in q plus any tuple $\langle \mathsf{p}_1, \ldots, \mathsf{p}_{n-1}, \mathsf{p}_n \rangle$ in p, so long as there is no tuple in q of the form $\langle \mathsf{p}_1, \ldots, \mathsf{p}_{n-1}, \mathsf{q}_n \rangle$ for some q_n. In the common case, p is a function containing the pair $\langle \mathsf{x}, \mathsf{y} \rangle$ and q is exactly the pair $\langle \mathsf{x}, \mathsf{z} \rangle$. The expression $\mathsf{p} \oplus \mathsf{q}$, in this case, evaluates to a function whose value is the same as p, except $\langle \mathsf{x}, \mathsf{y} \rangle$ has been replaced with $\langle \mathsf{x}, \mathsf{z} \rangle$. For example, consider the following expression:

$$\mathsf{email} \oplus \mathsf{newUser} \rightarrow \mathsf{newEmail}$$

Suppose again that newUser is `Greg`, but that email currently maps `Greg` to `gdennis@mit.edu` and that newEmail is now `drgreg@phd.com`. In this case, the expression evaluates to the relation that is the same as email, except it contains the tuple <`Greg`, `drgreg@phd.com`> in place of <`Greg`, `gdennis@mit.edu`>.

Boolean expressions can be formed by comparing expressions to one another using the equality ($=$), inequality ($\neq$), and subset ($\subseteq$) operators. Statement 1 in register, for example, tests whether the following expression is true:

$$\mathsf{newEmail} \subseteq \mathsf{User.email}$$

It is true if the singleton set newEmail is a subset of the emails currently taken by users, i.e., it tests whether newEmail is a fresh email.

Boolean expressions can also be formed with the standard logical connectives including ($\wedge$), or ($\vee$), not ($\neg$), and implies ($\Rightarrow$), and by universal ($\forall$) and existential ($\exists$) quantification. For example, if we add two more local variables, u1 and u2 of type User to the web registration program, then we can express the constraint that no two users have the same email:

$$\forall\ \mathsf{u1}, \mathsf{u2} \mid \mathsf{u1} \neq \mathsf{u2} \Rightarrow \mathsf{u1.email} \neq \mathsf{u2.email}$$

Quantified variables are implicitly quantified over their type, so this is a quantification of all singletons u1 and u2 drawn from the domain User.

A Contrast with Relational Logic

Although the FIR expression grammar is very similar to and inspired by the relational logic of Alloy, there is a key difference between the two. Unlike FIR, relational logic maintains a hard distinction between relational expressions, Boolean formulas, and integer expressions. In the logic, one cannot apply a relational operator, like union ($\cup$) or cross product ($\rightarrow$), to Booleans or integers, e.g. one cannot take the cross product of *true* and *false* or the union of *1* and *2*. Nor can one apply formula or integer operators to relational expressions, e.g. even if sets a and b each contain exactly one integer, one cannot take the sum of a and b.

The reason for this restriction is that the cardinality of (number of tuples in) a logic expression cannot be statically known in advance of solving. If integer addition could be applied to two sets, therefore, it would need to be well-defined in the case where either or both of those sets contained no elements or multiple elements. Rather than adopt an arbitrary meaning for integer and Boolean operators on sets, it maintains the distinction between relational, Boolean, and integer expressions, and disallows operators for one kind of expression from being applied to either of the two other kinds.

Due to a different set of priorities, FIR offers an alternative solution to this problem. FIR is meant to be a convenient intermediate representation into which high-level object-oriented languages can be translated, and programs in these languages include expressions like the following:

```
x.f && (x.g > 0)
```

where `f` is a field of type Boolean and `g` is a field of type integer in `T`. It is convenient for translations to FIR, like the Java translation to FIR discussed in Chapter 5, to translate this expression into the following FIR expression:

$$\mathsf{x.f} \wedge (\mathsf{x.g} > 0)$$

where f is a binary relation whose range is Boolean and g is a binary relation whose range is Integer. In contrast, the expression $(x.f \wedge x.g > 0)$ in relational logic would not be well-formed, because $x.f$ and $x.g$ are relational expressions to which Boolean operators like *and* ($\wedge$) and integer operators like *greater-than* ($>$) cannot be applied.

When the expression $(\mathsf{x.f} \wedge \mathsf{x.g} > 0)$ has been translated from the expression `(x.f && x.g > 0)` in code, x will be a singleton, f and g will be functions, and x.f and x.g will therefore be singletons. Nevertheless, in general Forge does not know the source of the FIR code, and x.f and x.g cannot statically be determined to be singletons in advance. As a result, to allow for expressions like $(\mathsf{x.f} \wedge \mathsf{x.g} > 0)$, FIR must give meaning to Boolean and integer operators when applied to sets.

Applying an integer operator to two sets means: take the sum of the integers in each operand and apply the standard meaning of the operator to those sums. For example, {3, 4} - {} evaluates to 7:

$$\{3, 4\} - \{\} = (3 + 4) - 0 = 7$$

Applying a Boolean operator to two sets means: take the disjunction (the Boolean sum) of each set and apply the operator to those two disjunctions. For example, {true, false} $\wedge$ {} evaluates to false:

$$\{\mathsf{true}, \mathsf{false}\} \wedge \{\} \;=\; (\mathsf{true} \vee \mathsf{false}) \wedge \mathsf{false} \;=\; \mathsf{true} \wedge \mathsf{false} \;=\; \mathsf{false}$$

These semantics of Boolean and integer operations could surprise a user who is coding directly in the intermediate representation. But, in the normal use case, the expressions are generated mechanically from high-level code, where the language dictates the operands be singletons, in which case the Boolean and integer operations have their standard meaning.

2.1.2 FIR Procedures

The signature of a FIR procedure consists of an identifier that names the procedure, followed by any number of input and output parameters given as lists of local variables. The signature of the register procedure, for example, declares that it takes one input, newEmail, and returns one output, newUser:

```
proc register (newEmail) : (newUser)
```

The setUniqueId procedure takes one input, newUser and has no outputs:

```
proc setUniqueId (newUser) : ()
```

The body of the procedures consist of a control-flow graph of program statements. There are no "return" statements in FIR code; instead, the final values of all output parameters are visible to a procedure's caller.

FIR Statements

There are three kinds of statements that appear in FIR control-flow graphs: *branch*, *update*, and *exit*. Every control-flow graph in FIR contains a single exit statement, marking the end of its execution. The first statement of the register procedure is a branch statement:

```
1  if newEmail ⊆ User.email
2     newUser := Error
   else
3     newUser := new User
```

Branch statements consist of a Boolean condition, in this case (newEmail $\subseteq$ User.email), and two successor statements in the control-flow graph. As explained, this condition is true if newUser is the email of an existing user. If the condition is true, control proceeds to the "true" successor; here that is Statement 2. Otherwise, control proceeds to the "false" successor, here Statement 3.

Update statements are the only statements that may modify the program state. There are four kinds of update statements: *assign*, *create*, *call*, and *spec*. Assign statements take the form v := e and assign the value of the expression e to the variable v. In the register procedure, there are two assignment statements:

```
2   newUser := Error
```

and

```
4   email := email ∪ newUser→newEmail
```

Statement 2 sets newUser to the Error literal. Statement 4 updates the value of email so that it contains a mapping from newUser to newEmail.

A create statement takes the form v := **new** D and dynamically allocates a new atom from the user-defined domain D and assigns it to v. This statement has the additional side-effect of adding the newly allocated atom to D. The register procedure contains a single create statement:

```
3   newUser := new User
```

It creates a new atom from the User domain and assigns it to newUser.

Call statements invoke procedures. To call a procedure p, it takes the general form $v_1, v_2, \ldots, v_n := p(a_1, a_2, \ldots, a_n)$, where $a_1 \ldots a_n$ are arguments to the procedure and $v_1 \ldots v_n$ are variables to which the output parameters of the procedure are assigned when the call returns. Consider, for example, the following swap procedure:

```
proc swap(x, y) : (x2, y2)
    x2 := y
    y2 := x
```

It could be called as follows to swap the values of a and b:

```
a, b := swap(a, b)
```

The final kind of update statement, a *specification statement*, is discussed in the next section.

The Specification Statement

An idea originating in refinement calculus [7, 65, 66, 67], the *specification statement* is a declarative specification that is embedded in otherwise imperative code. A specification statement first lists the frame condition (the variables that may be modified by the statement), followed by a Boolean expression that the statement ensures is true (by modifying at most the variables in the frame condition):

```
v1, v2 ... vn := spec (expr)
```

The expression relates the old and new values of the modified variables, referring to the old value of a variable v with the expression v_{old}, and the new values are chosen non-deterministically. For example, the following specification statement assigns to x non-deterministically so that its value is greater than its previous value:

$$x := \textbf{spec}\ (x > x_{old})$$

The first statement of the setUniqueId procedure is another example of a specification statement:

$$id := \textbf{spec}(\exists\, newId \mid (id = id_{old} \oplus newUser \rightarrow newId) \wedge \neg(newId \subseteq User.id_{old}))$$

It modifies the value of id to be equal to its previous value, id_{old}, overridden with a mapping from newUser to newId, where newId is some integer that is not currently the id of another user.

An important application of specification statements in the Forge framework is to enable modular analysis of high-level programs. As described in greater detail in Chapter 5, the JForge front-end does not examine the implementations of methods called by the method under analysis. Instead, JForge translates the specifications of those called methods into specification statements, and it uses these statements of their specification in place of their implementation. By doing so, JForge allows the implementation of the called methods to change, so long as its specification does not, without requiring a new analysis of the methods that call it. Also, since the specification is usually more compact and sometimes an abstraction of the code, using it in place of the implementation tends to greatly improve performance.

Many other intermediate representations, including that used by ESC/Java [37], offer equivalent support for specification statements in the form of *assume* and *havoc* statements. A *havoc* statement specifies a set of variables whose values may change non-deterministically. The FIR equivalent would be a specification statement with those variables on the left-hand side a right-hand side of true. An *assume* statement coerces a particular formula to be true; its FIR equivalent would be a specification statement with that formula on the right-hand side and an empty list of variables on the left. In general, a FIR specification statement of the following form:

$$v_1, v_2 \ldots v_n := \textbf{spec}\ (expr)$$

can be equivalently expressed in these other representations as

havoc $v_1, v_2 \ldots v_n$
assume expr

We prefer the specification statement because it is more concise, and because we can also use it to represent the specification *of* a procedure.

Infeasible Specifications

Specification statements should be used with some caution, however. The need for caution is due to the possibility of *infeasibility*; that is, there may exist *no* assignment to the variables on the left-hand side of a specification statement that makes its expression true. Consider if, for example, the subscript of the second occurrence of id_{old} were left out of the specification statement in register:

$$id := \textbf{spec}(\exists\, newId \mid (id = id_{old} \oplus newUser \rightarrow newId) \wedge \neg(newId \subseteq User.id))$$

Now the second constraint, $\neg(\mathsf{newId} \subseteq \mathsf{User.id})$, says that newId must not be the id of a user *in the post state*, which contradicts the first constraint, $(\mathsf{id} = \mathsf{id}_{old} \oplus \mathsf{newUser} \rightarrow \mathsf{newId})$, that newId be the id of newUser. So there is no assignment to id such that the expression is satisfied.

If there is no assignment to the variables in the frame condition that satisfies the the specification statement, the statement is said to be "infeasible." When we changed id_{old} to id, we created a specification statement that is infeasible on every execution of the code, but sometimes a specification statement in infeasible on some execution but not others. For example, the following specification statement that finds the square root of n is infeasible when n is negative:

$$\mathsf{r} := \mathbf{spec}(\mathsf{r} \times \mathsf{r} = \mathsf{n})$$

Semantically, an execution that encounters an infeasible specification statement results in a *miracle* [31, 65, 67, 70]. An analysis of such an infeasible execution will find it capable of ("miraculously") satisfying *any* post-condition, even the post-condition *false*. Consider, for example, the following piece of code:

$$\mathsf{n} := \text{-}1$$
$$\mathsf{r} := \mathbf{spec}(\mathsf{r} \times \mathsf{r} = \mathsf{n})$$

Does this code satisfy the post-condition $(\mathsf{r} \times \mathsf{r} = \mathsf{n})$? Yes, but since every execution of the code is infeasible — results in a miracle — it also satisfies the post-condition $(\mathsf{r} \times \mathsf{r} \neq \mathsf{n})$ and even $(\mathsf{r} \neq \mathsf{r})$. It satisfies anything and everything.

Miracles are certainly problematic when they arise accidentally. Because infeasible executions vacuously satisfy any post-condition, an analysis of code that contains an infeasible specification statement will effectively leave those executions unexplored. Indeed, it is not very useful to a user to find that all feasible executions of their code satisfy their post-condition, when in fact no feasible executions exist.

Due to their ability to satisfy any post-condition, Dijkstra recommended forbidding miracles, and thus his *Law of the Excluded Miracle* [31]. While the Forge framework in unable to automatically exclude miracles, its coverage metric, described in Chapter 4, can help a user detect them. Plus, as Morgan [65] and others have found, miracles have practical applications, too. For example, an infeasible specification may be included in a procedure *deliberately* to cause a subsequent analysis from exploring some set of executions. Loop unrolling, for instance, a process discussed below in Section 2-4, uses an infeasible specification statement for the explicit purpose of rendering executions infeasible if they exceed the unrolling limit.

2.2 Transforming FIR

The bounded verification analysis, explained in the following Chapters 3 and 4, accepts only a restricted subset of FIR that cannot contain loops or procedure calls. To ensure FIR code always meets this restriction, Forge performs two transformations — one that unrolls loops and the other that inlines procedure calls — to a FIR procedure before it is analyzed. These transformations are discussed below.

In addition to loop unrolling and call inlining, a client of the framework may additionally configure Forge to apply any number of custom transformations prior to analysis. For example, when JForge is the client, it configures Forge to apply a custom transformation that utilizes a dataflow analysis to find and eliminate logically infeasible branches.

Since the Forge analysis operates on the transformed FIR procedure, the trace and coverage information it computes is initially in terms of that transformed code, not the original. To convert the trace and coverage into terms of the original procedure, all transformations record a mapping from the statements in the transformed procedure to their corresponding source statements in the original. With this mapping, Forge automatically reconstructs the trace and coverage information of the original procedure from the results of analyzing the transformed one.

The basics of loop unrolling and call inlining are described in the next sections.

2.2.1 Unrolling Loops

This transformation unrolls each loop in the procedure according to a user-specified limit on the number of consecutive executions of the loop body. The transformation begins by unrolling all of the top-level (outer-most) loops in the procedure. If loops in the procedure remain, the transformation is applied again to unroll all the loops now at the top-level — loop that were formerly inner loops. This process is repeatedly applied until no loops remain. Note that this transformation will unroll an inner loop anew for every unrolling of the outer loop. That is, if the user requests m unrollings of a procedure that contains one outer loop and one inner loop, then the body of the inner loop will effectively be replicated $m \times m$ times.

To detect a top-level loop, our algorithm performs a depth-first search of the control-flow graph and checks whether each statement it encounters is the head of a loop, using the same criterion as Tarjan's interval-finding algorithm [81]. For every statement s in the control-flow graph of a procedure, Tarjan's algorithm defines s to be the head of a loop if there exists a back-edge in the control-flow graph whose target is s. Any nodes that can reach s via that back-edge are considered part of the loop body. Consider, for example, the contains procedure in Figure 2-4, which checks when an integer value is contained in a list. Tarjan's algorithm identifies Statement 1 in the procedure as a the head of a loop, because it is the target of the back-edge from Statement 3, and it identifies Statements 2 and 3 as the loop body.

A loop is unrolled by replicating its body for the specified number of unrollings. If there are inner loops within the body, they are not unrolled yet, but are replicated as the rest of the body was. They will be unrolled on subsequent passes of the algorithm. To illustrate the unrolling, Figure 2-5 shows the CFG of the contains procedure unrolled twice. To ensure that every execution exits the loop with the loop guard condition false, after the last statement of the last unrolling, the algorithm inserts a specification statement whose Boolean condition is the negation of the loop guard and whose frame condition is empty. In Figure 2-5, this is Statement 9.

```
domain List
global next: List→List, global value: List→Integer
local list: List, local val: Integer, local result: Boolean

proc contains(list, val) : (result)
1   while (list ≠ ∅) {
2      if (list.value ≠ val) {
3         list := list.next
       } else {
4         result := true
          exit
       }
    }
5   result := false
    exit
```

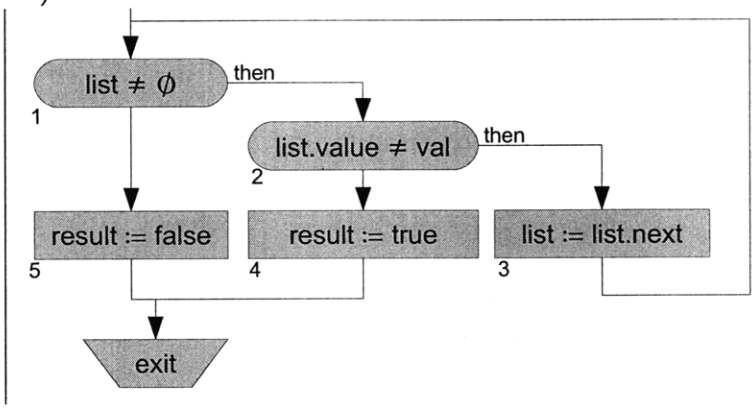


Figure 2-4: List containment procedure in FIR

```
1   if (list ≠ ∅) {
2      if (list.value ≠ val) {
3         list := list.next
6         if (list ≠ ∅) {
7            if (list.value ≠ val) {
8               list := list.next
9               spec(list := ∅)
                goto 5
             } else {
                goto 4
             }
          }
       } else {
4         result := true
          exit
       }
    }
5   result := false
    exit
```

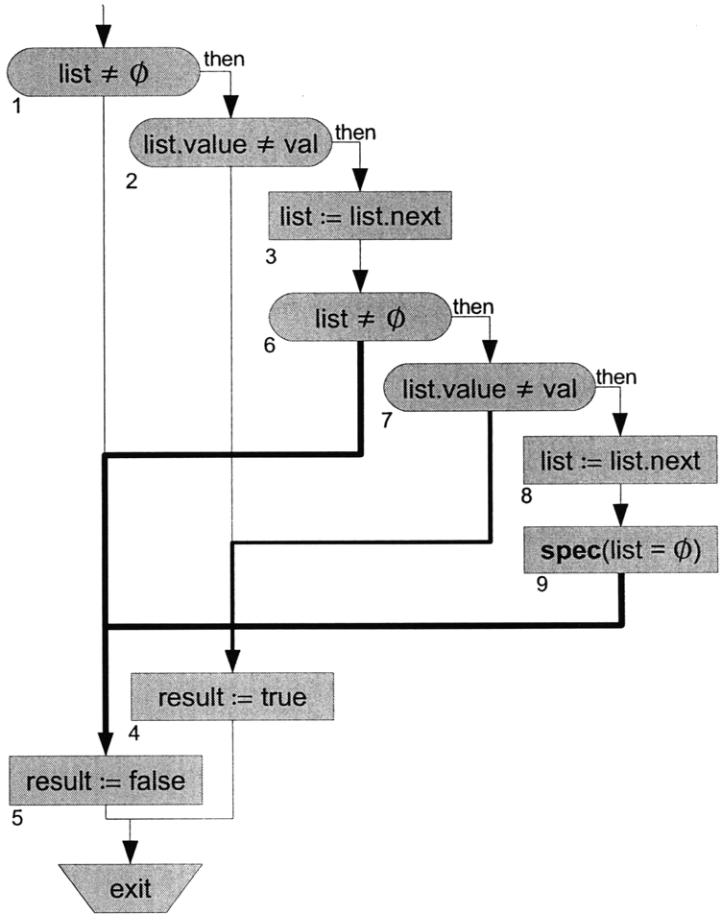


Figure 2-5: The contains procedure unrolled twice. Gotos indicate aliased statements. The thickness of the lines is used only to disambiguate line crossings.

If our algorithm finds a top-level loop that is the target of multiple back-edges, it rejects the procedure (throws a runtime exception), and the Forge analysis cannot be applied. Nevertheless, the algorithm is sufficient for unrolling CFGs generated from any structured code, including arbitrary combinations of loops, inner loops, if-statements, and break and continue statements. Unrestricted gotos could cause the unrolling to fail; but many languages, including Java and C, prohibit their use.

2.2.2 Inlining Procedure Calls

This transformation inlines the bodies of all called procedures into the control-flow graphs of their callers. Our example will be the following area procedure that calculates the area between two Cartesian points:

```
proc area(x, y, x2, y2) : (a)
    width := abs(x - x2)
    height := abs(y - y2)
    a := width × height
```

```
proc abs(x) : (a)
    if (x < 0) then a := -x else a := x
```

This section will show how the two calls to the abs procedure are inlined.

Prior to inlining a called procedure, all the local variables in the procedure, including its input and output parameters, are renamed to fresh local variables in order to avoid name conflicts:

```
proc abs(x') : (a')
    if (x' < 0) then a' := -x' else a' := x'
```

To inline a procedure call, the call statement is replaced with the following statements, in this order: (1) statements that assign the arguments of the call to the renamed input parameters of the called procedure; (2) statements that comprise the body of the renamed called procedure; and (3) statements that assign the output parameters to the variables that were assigned the result of the call. Thus, inlining the first call to abs into the area procedure produces the following result:

```
proc area(x, y, x2, y2) : (a)
    x' := x - x2
    if (x' < 0) then a' := -x' else a' := x'
    width := a'
    height := abs(y - y2)
    a := width × height
```

To inline the second call, the local variables in abs are again given fresh names:

```
proc abs(x'') : (a'')
    if (x'' < 0) then a'' := -x'' else a'' := x''
```

And the result of the second inlining looks as follows:

```
proc area(x, y, x2, y2) : (a)
    x' := x - x2
    if (x' < 0) then a' := -x' else a' := x'
    width := a'
    x'' := y - y2
    if (x'' < 0) then a'' := -x'' else a'' := x''
    height := a''
    a := width × height
```

If a call is recursive, that recursion is unrolled according to a client-specified limit, similar to loop unrolling.

2.3 Chapter Summary

This chapter presented the Forge Intermediate Representation (FIR). It covered the structure and semantics of the FIR, including its differences from relational logic and the uses and ramifications of specification statements in FIR. It also showed how Forge unrolls loops and inlines procedure calls that appear in FIR code. Once its loops are unrolled and calls inlined, a FIR procedure is ready for the bounded verification analysis presented in the next chapter.

Despite the richness of FIR's relational syntax, there are still areas in which it lacks some of the expressive available in high-level languages. For example, FIR lacks support for real numbers and real number arithmetic. While it would not be particularly difficult to add support for real numbers to FIR itself, we currently have no way of analyzing code that includes them, because the underlying Kodkod model finder upon which the tool relies provides no such support. Another area where FIR lacks expressiveness is in concurrency: it has no constructs for forking or joining threads or acquiring or releasing locks. In this area, however, we do not have Kodkod to blame. There is prior work on modular, formal verification of concurrent systems, and it should be possible for us to learn from this work in the future and apply our bounded verification to these systems as well.

Chapter 3

Bounded Verification

The bounded verification analysis takes as input a procedure in the Forge Intermediate Representation, a specification of the procedure provided in the form of a FIR specification statement, and a bound on the analysis, and it exhaustively searches within the bound for a trace of the procedure that violates the specification.

The bound provided to the analysis consists of the following:

- a number of times to unroll each loop and recursive call;
- a bitwidth to limit the range of integers in the Integer domain; and
- a *scope* on each user-defined domain in the program, i.e., a limit on the number of instances of that domain that may exist in any heap during execution.

Each of these limits results in under-approximation, eliminating, never adding, possible behaviors. Thus, any counterexample generated will be valid — either demonstrating a defect in the code or a flaw in the specification; and if a counterexample exists within the bound, one will be found, though defects that require a larger bound will be missed. (Although, as discussed in Section 5.1.3, the translation from a high-level programming language to FIR is free to use optimizations that undermine these soundness and completeness properties.)

The bounded verification begins by applying transformations to the FIR procedure under analysis. Loops are unrolled for the specified number of iterations and calls are inlined (with recursion unrolled for the same number of iterations), using the techniques previously described (Section 2.2).

The analysis then builds a formula in a relational logic that is true exactly when there is a trace of the transformed procedure that does not satisfy the specification. The construction of this formula uses a symbolic execution technique that traverses each branch in the code, building a symbolic relational expression for every variable at each program point.

The chosen bound and the formula are handed to the Kodkod model finder to solve. If Kodkod finds a solution, the analysis translates that solution into a counterexample trace of the transformed FIR procedure, and then finally to a trace of the original FIR procedure.

3.1 Model Finding with Relational Logic

The basic idea underlying the analysis is as follows. From the transformed FIR procedure (free of loops and procedure calls), the analysis automatically obtains a formula $P(s, s')$ in relational logic that constrains the relationship between a pre-state s and a post-state s' and that holds whenever an execution exists from s that terminates in s'.

A second formula $\psi(s, s')$ is obtained from a user-provided specification. Using P and ψ, the analysis checks the validity of the *correctness claim*, a formula that is true exactly when every trace of the procedure satisfies the specification:

$$\forall s, s' \mid P(s, s') \Rightarrow \psi(s, s')$$

However, since states s and s' are vectors of relations, solving the quantification over those states in the correctness claim requires enumerating all possible values of each relation. We call this a *higher-order* quantification because it amounts to quantifying over every value in a powerset, the size of which will be exponential in the scope on the analysis, and this exponential explosion makes the analysis intractable. To achieve a tractable analysis, the correctness claim is first negated to obtain the *refutation formula* which is true exactly for those executions that are possible but that violate the specification:

$$\exists s, s' \mid P(s, s') \wedge \neg\psi(s, s')$$

A solution to the refutation formula is a *counterexample* to the correctness claim.

The refutation formula still contains a higher-order quantification, but the existential quantifier can be eliminated by Skolemization. Skolemization turns the quantified variables s and s' into free variables, yielding a first-order formula that is equisatisfiable to the original[1]:

$$P(s, s') \wedge \neg\psi(s, s')$$

Forge invokes the Kodkod model finder to solve this Skolemized refutation formula, solutions to which witness traces of the procedure that violate the specification.

3.1.1 Using Kodkod

Kodkod translates the formula and the bound it is given into a Boolean satisfiability (SAT) problem, and invokes an off-the-shelf SAT solver on it. If the SAT solver finds a solution to the problem, Kodkod translates that SAT solution into a solution to the relational logic formula, and Forge translates that logic solution into a counterexample trace of the original FIR procedure. Kodkod is the result of many years of research into solving relational logic problems efficiently by an encoding to SAT. It incorporates compact sparse matrix representations and novel techniques for sharing detection and symmetry breaking. [84].

[1]The Skolemized refutation formula is first-order so long as the formulas P and $\neg\psi$ do not contain higher-order quantifiers themselves, which is prohibited by the intermediate representation.

An abstract syntax for the relational logic accepted by the Kodkod model finder is given in Figure 3-1. The logic is a core subset of the Alloy modeling language [48].

A logic *problem* consists of a *universe* declaration, a set of *relation declarations* and a set of *formulas* in which the declared relations appear as free variables. The universe is a finite set of *atoms* from which solutions to the problem will be drawn. The relations are declared with an arity, a lower bound, and an upper bound. The lower and upper bounds are *constants* — sets of tuples drawn from atoms in the universe. The upper bound is the set of tuples that *may* appear in the relation, and the lower bound is the set of tuples that *must* appear in the relation. The lower bounds of the declared relations are collectively referred to as the problem's *partial instance.* A solution to the problem binds each declared relational variable to a constant within that relation's bounds such that every formula is satisfied.

The *formula* and *expr* syntactic productions define a relational logic with transitive closure, first order quantifiers, and logical connectives. As illustrated by the *intexpr* production, the logic additionally supports integer expressions, including arithmetic and bitwise operators. An *intexpr* may be cast to an *expr* using the *int2expr* function. The *sum* function yields the integer that is the sum of all the integers in a set; if the set is singleton of one integer, *sum* can be regarded as a "cast" from an *expr* to an *intexpr*.

The following logic problem formulates the task of assigning teachers to classes for a semester. The problem involves three teachers (*Smith*, *Jones*, *Brown*) that will be assigned to teach four classes (*Math*, *History*, *English*, *Music*). Every class must have exactly one teacher and every teacher must teach at least one but less than three classes. Also, teacher *Smith* must always teach the *Math* class.

problem ::= *universe relDecl* formula**

universe ::= {*atom**}
relDecl ::= *relation* :$_{arity}$ [*constant*, *constant*]

constant ::= {*tuple**}
tuple ::= ⟨*atom**⟩

arity ::= positive integer
relation ::= identifier
atom ::= identifier

formula ::=

expr ⊂ *expr*	subset
| *expr* = *expr*	equality
| **some** *expr*	non-empty
| **one** *expr*	exactly one
| **lone** *expr*	empty or one
| **no** *expr*	empty
| ¬*formula*	negation
| *formula* ∧ *formula*	conjunction
| *formula* ∨ *formula*	disjunction
| *formula* ⇒ *formula*	implication
| ∀*varDecls* | *formula*	universal
| ∃*varDecls* | *formula*	existential
| *intexpr* {= | > | <} *intexpr*	int comparison

expr ::=

var	variable
| ~*expr*	transpose
| ^*expr*	closure
| *expr* ∪ *expr*	union
| *expr* ∩ *expr*	intersection
| *expr* \ *expr*	difference
| *expr*.*expr*	join
| *expr* → *expr*	product
| *formula* ? *expr* : *expr*	conditional
| {*varDecls* | *formula*}	comprehension
| π(*expr*, *intexpr**)	projection
| **int2expr**(*intexpr*)	int cast
| **univ**	universe

intexpr ::=

integer	literal
| #*expr*	cardinality
| **sum**(*e*)	sum
| Σ*varDecls* | *Expr*	summation
| *intexpr* {+| − | × |÷} *intexpr*	arithmetic
| *intexpr* {| | & | ^} *intexpr*	bitwise ops
| *intexpr* {≪ | ≫ | ⋙} *intexpr*	bit shifts

varDecls ::= (*variable* : *expr*)*
variable ::= *identifier*

Figure 3-1: Relational Logic

0 $\{Smith, Jones, Brown, Math, History, English, Music\}$

1 $Teacher :_1 [\{\}, \{\langle Smith\rangle, \langle Jones\rangle, \langle Brown\rangle\}]$
2 $Class :_1 [\{\langle Math\rangle, \langle History\rangle, \langle English\rangle, \langle Music\rangle\},$
$\{\langle Math\rangle, \langle History\rangle, \langle English\rangle, \langle Music\rangle\}]$
3 $teach :_2 [\{\langle Smith, Math\rangle\},$
$\{\langle Smith, Math\rangle, \langle Smith, History\rangle, \langle Smith, English\rangle, \langle Smith, Music\rangle,$
$\langle Jones, Math\rangle, \langle Jones, History\rangle, \langle Jones, English\rangle, \langle Jones, Music\rangle,$
$\langle Brown, Math\rangle, \langle Brown, History\rangle, \langle Brown, English\rangle, \langle Brown, Music\rangle\}]$

4 $teach \subseteq (Teacher \rightarrow Class)$
5 $\forall c : Class \mid \mathbf{one}\ teach.c$
6 $\forall t : Teacher \mid \mathbf{some}\ t.teach \wedge \#t.teach < 3$

The problem declares a universe of 7 atoms (Line 0) and three relations (Lines 1-3). The *Teacher* relation (Line 1) has an arity of 1 (it is a set) and represents the set of teachers employed for the semester; it is bounded below by the empty set and bounded above by the constant containing all the atoms that represent teachers. The *Class* relation (Line 2) is bounded both above and below by the atoms representing classes, so the set of classes taught is fixed to those exact four. The *teach* relation is binary. It is bounded below by the constant containing the tuple $\langle Smith, Math\rangle$ — to ensure *Smith* teaches *Math* — and above by the cross product of all the teacher and class atoms.

The problem consists of three constraints. According to the first constraint (Line 4), *teach* is a subset of of the cross product of the *Teacher* and *Class* sets, which means the *teach* relation only assigns the teachers for the semester to classes being taught. The second constraint (Line 5) says every class is taught by exactly one teacher. The third constraint (Line 6) says every teacher must teach at least one but less than 3 classes.

Kodkod translates the logic problem, upper bounds, and partial instances into a Boolean formula and invokes a SAT solver to find its satisfying solutions. Kodkod's support for partial instances is one of its key advantages over the Alloy Analyzer, and one which our tool exploits in its analysis. Because partial instances reflect fixed parts of the solution that do not need to be discovered, they enable Kodkod to reduce the sizes of the Boolean formulas it generates.

Here is a solution that Kodkod might find to the problem:

$Teacher \mapsto \{\langle Smith\rangle, \langle Brown\rangle\}$
$Class \mapsto \{\langle Math\rangle, \langle History\rangle, \langle English\rangle, \langle Music\rangle\}$
$teach \mapsto \{\langle Smith, Math\rangle, \langle Smith, English\rangle, \langle Brown, History\rangle, \langle Brown, Music\rangle\}$

In the solution, *Smith* and *Brown* are the teachers for the semester. *Smith* teaches both *Math* and *English*, and *Brown* teaches *History* and *Music*.

3.2 Symbolic Execution

The FIR procedure under analysis is translated to relational logic by a symbolic execution [50]. The *state* of the symbolic execution at each program point consists of three pieces of information:

- a *relational declaration*, $\mathcal{D}$: a set of relations from the logic;
- a *path constraint*, $\mathcal{P}$: a set of formulas that must be true for an execution to be feasible up to the current program point; and
- a *symbolic environment*, $\mathcal{E}$: a mapping of program variables and user-defined domains to expressions in the logic for their current value.

The symbolic state at a program point encodes the set of feasible program states at that point. Consider a program point for which the symbolic execution produces the symbolic state $\langle \mathcal{D}, \mathcal{P}, \mathcal{E} \rangle$. If there exists a binding of the relations in $\mathcal{D}$ to constants such that the formulas in $\mathcal{P}$ are true, then the execution may feasibly reach that program point. Furthermore, if we look up in $\mathcal{E}$ the expression to which each program variable is bound and evaluate that expression against the binding, we would produce a binding of program variables to constants that is a feasible state of the program at that program point. The example in the next section will help elucidate this idea.

3.2.1 Small Example: Social Network

This section demonstrates the symbolic execution on a small example and illustrates the key ideas involved. It omits some details of the symbolic execution, but these ideas will be explained in later sections. The example program records friendships between people, as in a simple social network. The befriend procedure adds a friendship two persons p and q:[2]

```
domain Person
global friends: Person→Person
local p: Person, local q: Person

proc befriend(p, q) : ()
  friends := friends ∪ p→q ∪ q→p
  exit
```

The specification we will check this procedure against is the preservation of the following invariant:

$$\mathsf{friends} := \mathbf{spec}(\mathsf{friends}_{old} = {\sim}\mathsf{friends}_{old} \Rightarrow \mathsf{friends} = {\sim}\mathsf{friends})$$

which says that if the friendships are symmetric in the pre-state then they are symmetric in the post-state. The FIR expression ∼r is the relational transpose of r.

[2]Technically, the parameters p and q may in general be sets of persons.

Small Example: The Initial State

For the befriend procedure, the symbolic execution constructs the following initial symbolic state — initial relational declaration $\mathcal{D}_0$, initial path constraint $\mathcal{P}_0$, and initial symbolic environment $\mathcal{E}_0$ — to represent all possible initial program states to the procedure:

$$\begin{aligned}\mathcal{D}_0 &= \{Person_0, friends_0, p_0, q_0\}\\ \mathcal{P}_0 &= \{friends_0 \subseteq Person_0 \rightarrow Person_0, p_0 \subseteq Person_0, q_0 \subseteq Person_0\}\\ \mathcal{E}_0 &= \{\mathsf{Person} \mapsto Person_0, \mathsf{friends} \mapsto friends_0, \mathsf{p} \mapsto p_0, \mathsf{q} \mapsto q_0\}\end{aligned}$$

In this initial state, the domain Person, the global variable friends, and the input parameters p and q are each bound to fresh relations for their value. The initial path constraint says that the initial values for friends, p, and q contain only persons that exist in this initial state. Now consider any binding of the variables in $\mathcal{D}_0$ to constants such that the formulas in $\mathcal{P}_0$ are true. Here is one such binding:

$$\begin{aligned}Person_0 &= \{\langle \mathtt{P1}\rangle, \langle \mathtt{P2}\rangle, \langle \mathtt{P3}\rangle\}\\ friends_0 &= \{\langle \mathtt{P1}, \mathtt{P2}\rangle, \langle \mathtt{P2}, \mathtt{P1}\rangle, \langle \mathtt{P2}, \mathtt{P3}\rangle\}\\ p_0 &= \{\langle \mathtt{P1}\rangle\}\\ q_0 &= \{\langle \mathtt{P3}\rangle\}\end{aligned}$$

By evaluating the expression to which each variable is mapped in $\mathcal{E}_0$ against this binding, we derive a possible initial state to the program:

$$\begin{aligned}\mathsf{Person} &= \{\langle \mathtt{P1}\rangle, \langle \mathtt{P2}\rangle, \langle \mathtt{P3}\rangle\}\\ \mathsf{friends} &= \{\langle \mathtt{P1}, \mathtt{P2}\rangle, \langle \mathtt{P2}, \mathtt{P1}\rangle, \langle \mathtt{P2}, \mathtt{P3}\rangle\}\\ \mathsf{p} &= \{\langle \mathtt{P1}\rangle\}\\ \mathsf{q} &= \{\langle \mathtt{P3}\rangle\}\end{aligned}$$

The single symbolic state is encoding all such feasible initial program states. This particular program state does not satisfy the invariant, but that is not of concern at this point. The symbolic execution is conducted *independently* of any specification. The specification will be taken into account once the symbolic execution is complete.

Now consider the effect of leaving out the formulas in the initial path constraint. If they were left out, the initial symbolic state would also encode initial program states like the following:

$$\begin{aligned}\mathsf{Person} &= \{\langle \mathtt{P1}\rangle, \langle \mathtt{P2}\rangle\}\\ \mathsf{friends} &= \{\langle \mathtt{P1}, \mathtt{P3}\rangle, \langle \mathtt{P3}, \mathtt{P1}\rangle\}\\ \mathsf{p} &= \{\langle \mathtt{P2}\rangle\}\\ \mathsf{q} &= \{\langle \mathtt{P3}\rangle\}\end{aligned}$$

where the variables friends and q contain atoms representing persons that do not exist (at least not yet). Such program states are not feasible initial states to the procedure, and the formulas in the initial path constraint are needed to exclude them.

Small Example: Executing the Statements

Once the initial symbolic state is constructed, the body of the procedure is symbolically executed which, in the befriend procedure, consists of the following assignment:

$$\mathsf{friends} := \mathsf{friends} \cup \mathsf{p} \rightarrow \mathsf{q} \cup \mathsf{q} \rightarrow \mathsf{p}$$

To symbolically execute this assignment the symbolic execution, we first translate the FIR expression on the right-hand side into an expression in relational logic. Since the current symbolic environment $\mathcal{E}_0$ maps the FIR variables friends, p, and q to the expressions $friends_0$, p_0, and q_0, respectively, the right-hand side is straightforwardly translated to the following logic expression:

$$friends_0 \cup p_0 \rightarrow q_0 \cup q_0 \rightarrow p_0$$

The analysis can symbolically execute this assignment statement in two possible ways. The first way maps the assigned FIR variable, friends, to the logic expression $friends_0 \cup p_0 \rightarrow q_0 \cup q_0 \rightarrow p_0$, and does not modify the relational declaration or path constraint. This produces the following symbolic state, which being the final symbolic state of the procedure, we mark with the subscript f:

$$\begin{aligned}
\mathcal{D}_f &= \{Person_0, friends_0, p_0, q_0\} \\
\mathcal{P}_f &= \{friends_0 \subseteq Person_0 \rightarrow Person_0, p_0 \subseteq Person_0, q_0 \subseteq Person_0\} \\
\mathcal{E}_f &= \{\mathsf{Person} \mapsto Person_0,\ \mathsf{p} \mapsto p_0,\ \mathsf{q} \mapsto q_0, \\
&\qquad \mathsf{friends} \mapsto friends_0 \cup p_0 \rightarrow q_0 \cup q_0 \rightarrow p_0\ \}
\end{aligned}$$

The change to the environment is highlighted in gray.

The other, equivalent, symbolic execution of this statement adds a fresh relation, $friends_1$, to the relational declaration to store the value of friends after the statement. It constrains $friends_1$ to equal $friends_0 \cup p_0 \rightarrow q_0 \cup q_0 \rightarrow p_0$, and it maps friends to $friends_1$ in the symbolic environment:

$$\begin{aligned}
\mathcal{D}_f &= \{Person_0, friends_0, p_0, q_0,\ friends_1\ \} \\
\mathcal{P}_f &= \{friends_0 \subseteq Person_0 \rightarrow Person_0, p_0 \subseteq Person_0, q_0 \subseteq Person_0, \\
&\qquad friends_1 = friends_0 \cup p_0 \rightarrow q_0 \cup q_0 \rightarrow p_0\ \} \\
\mathcal{E}_f &= \{Person \mapsto Person_0,\ \mathsf{p} \mapsto p_0,\ \mathsf{q} \mapsto q_0,\ \mathsf{friends} \mapsto friends_1\ \}
\end{aligned}$$

The two ways to symbolically execute this assignment statement reflect two approaches to the symbolic execution generally. The first way was an example of the *inline strategy*. The inline strategy for symbolic execution tries to avoid, whenever possible, adding fresh relations to the symbolic state[3]. The second way was an example of the *constrain strategy*. The constrain strategy maps each modified variable to a fresh relation and constrains the value of that variable by adding new path constraints. These two strategies are equivalent in that they encode the same set of feasible program executions. The choice of strategy has performance implications for the performance of the Kodkod analysis (Section 6.3.1), and for the accuracy of the coverage metric (Chapter 4).

[3]Some statements require that fresh relations be declared, e.g. specification statements, because they allows variables to change non-deterministically.

Small Example: Building the Logic Problem

Once the symbolic execution of the procedure is complete, the analysis generates, from the provided specification, a problem in relational logic for the Kodkod model finder to solve. The specification for our befriend procedure is that the symmetry of the friendships is preserved:

$$\mathsf{friends} := \textbf{spec}(\mathsf{friends}_{old} = \sim\mathsf{friends}_{old} \Rightarrow \mathsf{friends} = \sim\mathsf{friends})$$

From this specification, the analysis creates a logic formula ψ by replacing all the occurrences of $\mathsf{friends}_{old}$ with the expression to which friends is bound in the initial symbolic environment (in our example, $friends_0$) and replacing all the occurrences of friends with its expression in the final environment (an expression which differs depending on the symbolic execution strategy).

If the inline strategy were used, the final environment would bind friends to the expression $(friends_0 \cup p_0 \rightarrow q_0 \cup q_0 \rightarrow p_0)$, so ψ would be:

$$\begin{aligned}&friends_0 = \sim friends_0 \Rightarrow \\ &(friends_0 \cup p_0 \rightarrow q_0 \cup q_0 \rightarrow p_0) = \sim(friends_0 \cup p_0 \rightarrow q_0 \cup q_0 \rightarrow p_0)\end{aligned}$$

If the constrain strategy were used, the final environment would bind friends to the expression $friends_1$, so ψ would be:

$$friends_0 = \sim friends_0 \Rightarrow friends_1 = \sim friends_1$$

Finally, the analysis asks Kodkod to find an assignment to the logic variables in the final relational declaration, $\mathcal{D}_f$, such that the formulas in $\mathcal{P}_f$ are true and ψ is not, i.e., a solution to the formula $\mathcal{P}_f \wedge \neg\psi$. With the inline strategy, this formula would be:

$$\begin{aligned}&(friends_0 \subseteq Person_0 \rightarrow Person_0) \wedge (p_0 \subseteq Person_0) \wedge (q_0 \subseteq Person_0) \wedge \\ &\neg(friends_0 = \sim friends_0 \Rightarrow \\ &\quad (friends_0 \cup p_0 \rightarrow q_0 \cup q_0 \rightarrow p_0) = \sim(friends_0 \cup p_0 \rightarrow q_0 \cup q_0 \rightarrow p_0))\end{aligned}$$

With the constrain strategy it would be:

$$\begin{aligned}&(friends_0 \subseteq Person_0 \rightarrow Person_0) \wedge (p_0 \subseteq Person_0) \wedge (q_0 \subseteq Person_0) \wedge \\ &(friends_1 = friends_0 \cup p_0 \rightarrow q_0 \cup q_0 \rightarrow p_0) \wedge \\ &\neg(friends_0 = \sim friends_0 \Rightarrow friends_1 = \sim friends_1)\end{aligned}$$

A solution to either formula witnesses a feasible execution of the procedure that violates the specification.

3.2.2 Building the Initial State

We will now explain the symbolic execution in detail. Given a FIR program, the symbolic execution begins by populating the initial relational declaration with several relational logic variables. To encode FIR literals, $\mathcal{D}_0$ contains the following relations:

- *true* and *false*, singletons to encode the true and false literals in FIR;
- *Integer*, the set containing all the integers in the specified bitwidth, to encode the value of the FIR Integer domain; and
- for every user-defined literal L in the FIR program, a corresponding unary relation L to encode its value.

For every user-defined domain M in the program, $\mathcal{D}_0$ contains a corresponding relation M_0, and the initial symbolic environment $\mathcal{E}_0$ maps M to M_0.

For every input parameter (to the procedure analysis) and global variable v, $\mathcal{D}_0$ contains a corresponding relation v_0, $\mathcal{E}_0$ maps v to v_0, and the initial path constraint, $\mathcal{P}_0$, contains a formula that restricts v_0 to be a subset (or sub-relation) of the initial value of its type. That is, if the type of v is a cross-product of domains $\mathsf{M}_1 \rightarrow \mathsf{M}_2 \rightarrow \ldots \rightarrow \mathsf{M}_n$, then $\mathcal{P}_0$ contains a constraint of the following form:

$$v_0 \subseteq M_{1_0} \rightarrow M_{2_0} \rightarrow \ldots \rightarrow M_{n_0}$$

where $M_{1_0}, M_{2_0} \ldots M_{n_0}$ are the relations to which the domains M_1, M_2, $\ldots \mathsf{M}_n$ are bound in the initial symbolic environment.

3.2.3 Translating Expressions

The symbolic execution frequently translates expressions in FIR into expressions in relational logic. This is mostly a straightforward task because, for nearly every operator in FIR, there is a corresponding operator in the relational logic with the same semantics. For example, if the FIR variables x and y are bound to the relations x and y in the symbolic environment, then the symbolic execution translates the FIR expression $(\mathsf{x} \cup \mathsf{y})$ to the logic expression $(x \cup y)$.

However, this otherwise straightforward task is slightly complicated by the difference discussed in Section 2.1.1 between the FIR expression grammar and the logic expression grammar. Recall that in FIR, one can write the following expression:

$$\mathsf{x.f} \wedge (\mathsf{x.g} > 0)$$

where x is a set, f is a binary relation whose range is Boolean, and g is a binary relation whose range is Integer. However, the identical expression in relational logic would not well-formed, because $x.f$ and $x.g$ are relational expressions to which Boolean operators like *and* ($\wedge$) and integer operators like *greater-than* ($>$) cannot be applied.

As Section 2.1.1 explained, the ability to apply Boolean and integer operators to relation-valued expressions requires that FIR give meaning to Boolean and integer operators when applied to non-singleton sets. In FIR, applying an integer operator

to two sets means: take the sum of the integers in each operand and apply the standard meaning of the operator to those sums. Thus, in general, the FIR expression $(\mathsf{a} \star \mathsf{b})$ is translated into the logic expression $(sum(a) \star sum(b))$, for every integer operator $\star$. Applying a Boolean operator to two sets means: take the disjunction (the Boolean sum) of each set and apply the operator to those two disjunction. Note that a disjunction of Booleans is true if and only if true is an element of that set. Thus, in general, the FIR expression $(\mathsf{a} \star \mathsf{b})$ is translated into the logic expression $((true \subseteq a) \star (true \subseteq b))$, for every Boolean operator $\star$.

We've discussed how the symbolic execution translates Boolean and integer operations on set-valued operands, but it must also be able to translate relational operations to the logic when the operands are Booleans or integers. Consider, for example, the FIR expression $(1 \cup 2)$ that takes the union of integers 1 and 2 and evaluates to the set containing exactly both integers. This union expression cannot be translated naively to the logic expression $(1 \cup 2)$, because union ($\cup$) is a relational operator that cannot be applied to integer expressions. The translation must essentially "cast" the integers to singleton sets before the union is applied. To cast an integer expression i to a set, the translation uses the logic's built-in *int2expr* function. Therefore, the FIR expression $(1 \cup 2)$ is translated into the logic expression $(int2expr(1) \cup int2expr(2))$. To cast a Boolean formula f to a set, the translation builds a conditional expression of the form $(f \ ? \ true : false)$, where *true* and *false* are the relations in $\mathcal{D}_0$ that correspond to the FIR literals true and false.

The translation of FIR expressions to logic expressions is given in Figure 3-2. It is defined in terms of three functions: $\mathcal{T}_R$ for translating FIR expressions to relational expressions in the logic, $\mathcal{T}_F$ for translating FIR expressions to Boolean formulas in the logic, and $\mathcal{T}_I$ for translating to integer expressions in the logic. As shown in the figure, the translation of a FIR expression depends upon the current symbolic environment.

For example, consider the symbolic environment that maps x to p and y to q:

$$\mathcal{E} = \{\mathsf{x} \mapsto p, \mathsf{y} \mapsto q\}$$

In this environment, translating the FIR expression $\mathsf{x} \cup \mathsf{y}$ to a relational expression in the logic yields $p \cup q$:

$$\mathcal{T}_R[\![\mathsf{x} \cup \mathsf{y}, \mathcal{E}]\!] = \mathcal{T}_R[\![\mathsf{x}, \mathcal{E}]\!] \cup \mathcal{T}_R[\![\mathsf{y}, \mathcal{E}]\!] = \mathcal{E}(\mathsf{x}) \cup \mathcal{E}(\mathsf{y}) = p \cup q$$

But translating the FIR expression $\mathsf{x} + \mathsf{y}$ in the environment to a relational expression in the logic involves "casting" back and forth between relations and integer in the logic to yield $int2expr(sum(p) + sum(q))$:

$$\begin{aligned}&\mathcal{T}_R[\![\mathsf{x} + \mathsf{y}, \mathcal{E}]\!] = \\ &int2expr(\mathcal{T}_I[\![\mathsf{x}, \mathcal{E}]\!] + \mathcal{T}_I[\![\mathsf{y}, \mathcal{E}]\!]) = int2expr(sum(\mathcal{T}_R[\![\mathsf{x}, \mathcal{E}]\!]) + sum(\mathcal{T}_R[\![\mathsf{y}, \mathcal{E}]\!])) = \\ &int2expr(sum(\mathcal{E}(\mathsf{x})) + sum(\mathcal{E}(\mathsf{y}))) = int2expr(sum(p) + sum(q))\end{aligned}$$

In addition to binding variables and user-defined domains to expressions, the symbolic environment, as shown in Figure 3-2, can also bind "old" variables to expressions. The environment will only bind old variables when the symbolic execution is translating FIR specifications to logic.

$Modifiable \equiv \mathsf{Variable} \cup \mathsf{UserDomain} \cup \{\mathsf{v}_{old} \mid \mathsf{v} \in \mathsf{Variable}\}$
$\mathcal{E} \in Env \equiv Modifiable \rightarrow expr$

$\mathcal{T}_R : \mathsf{Expr} \times Env \rightarrow expr$
$\mathcal{T}_F : \mathsf{Expr} \times Env \rightarrow formula$
$\mathcal{T}_I : \mathsf{Expr} \times Env \rightarrow intexpr$

$\mathcal{T}_R[\![\mathsf{true}, \mathcal{E}]\!] = true$
$\mathcal{T}_R[\![\mathsf{false}, \mathcal{E}]\!] = false$
$\mathcal{T}_R[\![\mathsf{i}, \mathcal{E}]\!] = int2expr(i)$, for an integer literal i
$\mathcal{T}_R[\![\mathsf{Boolean}, \mathcal{E}]\!] = true \cup false$
$\mathcal{T}_R[\![\mathsf{Integer}, \mathcal{E}]\!] = Integer$
$\mathcal{T}_R[\![\mathsf{L}, \mathcal{E}]\!] = L$, for an instance literal L
$\mathcal{T}_R[\![\mathsf{v}, \mathcal{E}]\!] = \mathcal{E}(\mathsf{v})$, for a variable, instance domain, or old variable v

$\mathcal{T}_R[\![\star\mathsf{e}, \mathcal{E}]\!] = \star\mathcal{T}_R[\![\mathsf{e}, \mathcal{E}]\!]$, for $\star$ in $\{\hat{},\sim\}$
$\mathcal{T}_R[\![\star\mathsf{e}, \mathcal{E}]\!] = form2expr(\star\mathcal{T}_R[\![\mathsf{e}, \mathcal{E}]\!])$, for $\star$ in $\{\mathsf{some},\mathsf{one},\mathsf{lone},\mathsf{no}\}$
$\mathcal{T}_R[\![\neg\mathsf{e}, \mathcal{E}]\!] = form2expr(\neg\mathcal{T}_F[\![\mathsf{e}, \mathcal{E}]\!])$
$\mathcal{T}_R[\![\star\mathsf{e}, \mathcal{E}]\!] = int2expr(\star\mathcal{T}_R[\![\mathsf{e}, \mathcal{E}]\!])$, for $\star$ in $\{\mathsf{sum},\#\}$

$\mathcal{T}_R[\![\mathsf{e}_1\star\mathsf{e}_2, \mathcal{E}]\!] = \mathcal{T}_R[\![\mathsf{e}_1, \mathcal{E}]\!] \star \mathcal{T}_R[\![\mathsf{e}_2, \mathcal{E}]\!]$, for $\star$ in $\{\cup, \cap, \setminus, \rightarrow, .\}$
$\mathcal{T}_R[\![\mathsf{e}_1\star\mathsf{e}_2, \mathcal{E}]\!] = form2expr(\mathcal{T}_R[\![\mathsf{e}_1, \mathcal{E}]\!] \star \mathcal{T}_R[\![\mathsf{e}_2, \mathcal{E}]\!])$, $\star$ in $\{=, \neq, \subseteq\}$
$\mathcal{T}_R[\![\mathsf{e}_1\star\mathsf{e}_2, \mathcal{E}]\!] = form2expr(\mathcal{T}_F[\![\mathsf{e}_1, \mathcal{E}]\!] \star \mathcal{T}_F[\![\mathsf{e}_2, \mathcal{E}]\!])$, $\star$ in $\{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$
$\mathcal{T}_R[\![\mathsf{e}_1\star\mathsf{e}_2, \mathcal{E}]\!] = form2expr(\mathcal{T}_I[\![\mathsf{e}_1, \mathcal{E}]\!] \star \mathcal{T}_I[\![\mathsf{e}_2, \mathcal{E}]\!])$, $\star$ in $\{>, <, \geq, \leq\}$
$\mathcal{T}_R[\![\mathsf{e}_1\star\mathsf{e}_2, \mathcal{E}]\!] = int2expr(\mathcal{T}_I[\![\mathsf{e}_1, \mathcal{E}]\!] \star \mathcal{T}_I[\![\mathsf{e}_2, \mathcal{E}]\!])$, $\star$ in $\{+, -, \times, \div, |, \&, \hat{}, \ll, \gg, \ggg\}$

$\mathcal{T}_R[\![\mathsf{e}_1 \oplus \mathsf{e}_2, \mathcal{E}]\!] = (\mathcal{T}_R[\![\mathsf{e}_1, \mathcal{E}]\!] \setminus (\pi(e, 1, \ldots, n-1) \rightarrow univ)) \cup e$
where $e = \mathcal{T}_R[\![\mathsf{e}_2, \mathcal{E}]\!]$, n is the arity of e, $\pi(e, 1, \ldots, n-1)$
is the relational projection over the first $n-1$ columns of e,
and $univ$ is the universal set of all atoms

$\mathcal{T}_R[\![\star\mathsf{v} \mid \mathsf{e}, \mathcal{E}]\!] = form2expr(\star var : \mathcal{T}_R[\![\mathsf{T}, \mathcal{E}]\!] \mid \mathcal{T}_F[\![\mathsf{e}, \mathcal{E}[\mathsf{v} \mapsto var]]\!])$
where $\star$ in $\{\forall, \exists\}$, var is a fresh logic variable, and T is the type of v

$\mathcal{T}_F[\![\mathsf{e}, \mathcal{E}]\!] = true \subseteq \mathcal{T}_R[\![\mathsf{e}, \mathcal{E}]\!]$
$\mathcal{T}_I[\![\mathsf{e}, \mathcal{E}]\!] = sum(\mathcal{T}_R[\![\mathsf{e}, \mathcal{E}]\!])$

$form2expr(f) = f \ ?\ true : false$
$int2expr(i) =$ the singleton set containing the integer i, a built-in function in Kodkod

Figure 3-2: Semantics of FIR Expression Translation

3.2.4 Formal Execution Rules

Each step of the symbolic execution takes as input a FIR statement and the current symbolic state (a relational declaration, path constraint, and symbolic environment) and yields a new symbolic state that reflects the effect of the statement. For assign, create, and branch statements, the symbolic execution may apply either a rule from the inline strategy or the constrain strategy to generate the new state. The formal symbolic execution rules are given in Figure 3-3 and are explained below.

Assignment Statement Rules

A FIR assignment statement has the following form:

$$\mathsf{v} := \mathsf{e}$$

Starting from the symbolic static $\langle \mathcal{D}, \mathcal{P}, \mathcal{E} \rangle$, the inline rule for an assignment statement does not alter the relational model, $\mathcal{D}$, or the path constraint, $\mathcal{P}$, but maps the assigned variable to the translation of the expression on the right-hand side, yielding the symbolic state:

$$\langle \mathcal{D}, \mathcal{P}, \mathcal{E}[\mathsf{v} \mapsto \mathcal{T}_R[\![\mathsf{e}, \mathcal{E}]\!]] \rangle$$

The constrain rule for an assignments statement introduces a fresh relation v' to store the new value of the assigned variable, constrains that relation to equal the translation of the right-hand side expression, and maps the assigned variable to the fresh relation, yielding the symbolic state:

$$\langle \mathcal{D} \cup \{v'\}, \mathcal{P} \cup \{v' = \mathcal{T}_R[\![\mathsf{e}, \mathcal{E}]\!]\}, \mathcal{E}[\mathsf{v} \mapsto v'] \rangle$$

Create Statement Rules

A FIR create statement has the following form:

$$\mathsf{v} := \textbf{new}\ \mathsf{M}$$

Let *currs* be $\mathcal{E}(\mathsf{M})$, an expression which evaluates to the set of atoms currently in the domain M. The inline strategy for create statements declares a fresh relation v' for the new value of v and constrains v' to be a singleton that is not currently in M. The symbolic environment after the statement maps v to v' and M to the union of the atoms currently in M and v':

$$\langle \mathcal{D} \cup \{v'\}, \mathcal{P} \cup \{\textbf{one}\ v', \neg(v' \subseteq \mathit{currs})\}, \mathcal{E}[\mathsf{v} \mapsto v',\ \mathsf{M} \mapsto (\mathit{currs} \cup v')] \rangle$$

The constrain strategy is similar, except it also declares a fresh relation, M' for the new value of the domain M, constrains M' to equal the union of *currs* and v', and maps M to M' in the symbolic environment:

$$\langle \mathcal{D} \cup \{v', M'\}, \mathcal{P} \cup \{\textbf{one}\ v', \neg(v' \subseteq \mathit{currs}), M' = \mathit{currs} \cup v'\}, \mathcal{E}[\mathsf{v} \mapsto v',\ \mathsf{M} \mapsto M'] \rangle$$

$\mathcal{D} \in Decl \equiv$ *set of relations*
$\mathcal{P} \in Path \equiv$ *set of formulas*
$\mathcal{E} \in Env \equiv Modifiable \rightarrow expr$
$\Longrightarrow$: $Stmt \times \langle Decl, Path, Env \rangle \rightarrow \langle Decl, Path, Env \rangle$

$$\frac{\mathsf{S}_1, \langle \mathcal{D}, \mathcal{P}, \mathcal{E} \rangle \Longrightarrow \langle \mathcal{D}_1, \mathcal{P}_1, \mathcal{E}_1 \rangle \quad \mathsf{S}_2, \langle \mathcal{D}_1, \mathcal{P}_1, \mathcal{E}_1 \rangle \Longrightarrow \langle \mathcal{D}_2, \mathcal{P}_2, \mathcal{E}_2 \rangle}{\mathsf{S}_1;\mathsf{S}_2, \langle \mathcal{D}, \mathcal{P}, \mathcal{E} \rangle \Longrightarrow \langle \mathcal{D}_2, \mathcal{P}_2, \mathcal{E}_2 \rangle}$$

$$\mathsf{v} := \mathsf{e}, \langle \mathcal{D}, \mathcal{P}, \mathcal{E} \rangle \Longrightarrow_I \langle \mathcal{D}, \mathcal{P}, \mathcal{E}[\mathsf{v} \mapsto \mathcal{T}_R[\![\mathsf{e}, \mathcal{E}]\!]] \rangle$$

$$\mathsf{v} := \mathsf{e}, \langle \mathcal{D}, \mathcal{P}, \mathcal{E} \rangle \Longrightarrow_C \langle \mathcal{D} \cup \{v'\}, \mathcal{P} \cup \{v' = \mathcal{T}_R[\![\mathsf{e}, \mathcal{E}]\!]\}, \mathcal{E}[\mathsf{v} \mapsto v'] \rangle$$

$$\frac{\mathit{currs} = \mathcal{E}(\mathsf{M})}{\begin{array}{c}\mathsf{v} := \textbf{new } \mathsf{M}, \langle \mathcal{D}, \mathcal{P}, \mathcal{E} \rangle \Longrightarrow_I \\ \langle \mathcal{D} \cup \{v'\}, \mathcal{P} \cup \{\textbf{one } v', \neg(v' \subseteq \mathit{currs})\}, \mathcal{E}[\mathsf{v} \mapsto v', \mathsf{M} \mapsto (\mathit{currs} \cup v')] \rangle\end{array}}$$

$$\frac{\mathit{currs} = \mathcal{E}(\mathsf{M})}{\begin{array}{c}\mathsf{v} := \textbf{new } \mathsf{M}, \langle \mathcal{D}, \mathcal{P}, \mathcal{E} \rangle \Longrightarrow_C \\ \langle \mathcal{D} \cup \{v', M'\}, \mathcal{P} \cup \{\textbf{one } v', \neg(v' \subseteq \mathit{currs}), M' = \mathit{currs} \cup v'\}, \mathcal{E}[\mathsf{v} \mapsto v', \mathsf{M} \mapsto M'] \rangle\end{array}}$$

$$\frac{\mathcal{E}_{new} = \mathcal{E}[\mathsf{v}_1 \mapsto v'_1, \ldots, \mathsf{v}_n \mapsto v'_n] \quad \mathcal{E}_{spec} = \mathcal{E}_{new}[\mathsf{v}_{1old} \mapsto \mathcal{E}(\mathsf{v}_1), \ldots, \mathsf{v}_{nold} \mapsto \mathcal{E}(\mathsf{v}_n)]}{\begin{array}{c}\mathsf{v}_1, \ldots, \mathsf{v}_n := \textbf{spec}(\mathsf{e}), \langle \mathcal{D}, \mathcal{P}, \mathcal{E} \rangle \Longrightarrow \\ \langle \mathcal{D} \cup \{v'_1, \ldots, v'_n\}, \mathcal{P} \cup \{\mathcal{T}_F[\![\mathsf{e}, \mathcal{E}_{spec}]\!], v'_1 \subseteq \mathcal{T}_R[\![\mathsf{T}_1, \mathcal{E}_{new}]\!], \ldots, v'_n \subseteq \mathcal{T}_R[\![\mathsf{T}_n, \mathcal{E}_{new}]\!]\}, \mathcal{E}_{new} \rangle\end{array}}$$

where $\mathsf{T}_1, \ldots, \mathsf{T}_n$ are the FIR types of $\mathsf{v}_1, \ldots, \mathsf{v}_n$, respectively.

$$\frac{\mathsf{S}_T, \langle \mathcal{D}, \mathcal{P}, \mathcal{E} \rangle \Longrightarrow \langle \mathcal{D}_T, \mathcal{P}_T, \mathcal{E}_T \rangle \quad \mathsf{S}_F, \langle \mathcal{D}, \mathcal{P}, \mathcal{E} \rangle \Longrightarrow \langle \mathcal{D}_F, \mathcal{P}_F, \mathcal{E}_F \rangle}{\begin{array}{c}\textbf{if } \mathsf{e} \textbf{ then } \mathsf{S}_T \textbf{ else } \mathsf{S}_F, \langle \mathcal{D}, \mathcal{P}, \mathcal{E} \rangle \Longrightarrow_I \\ \langle \mathcal{D}_B, \mathcal{P}_B, \mathcal{E}_B[\{\mathsf{v} \mapsto (\mathit{cond}\ ?\ \mathcal{E}_T(\mathsf{v}) : \mathcal{E}_F(\mathsf{v})) \mid \mathsf{v} \in \mathcal{V}_B\}] \rangle\end{array}}$$

$$\frac{\mathsf{S}_T, \langle \mathcal{D}, \mathcal{P}, \mathcal{E} \rangle \Longrightarrow \langle \mathcal{D}_T, \mathcal{P}_T, \mathcal{E}_T \rangle \quad \mathsf{S}_F, \langle \mathcal{D}, \mathcal{P}, \mathcal{E} \rangle \Longrightarrow \langle \mathcal{D}_F, \mathcal{P}_F, \mathcal{E}_F \rangle}{\begin{array}{c}\textbf{if } \mathsf{e} \textbf{ then } \mathsf{S}_T \textbf{ else } \mathsf{S}_F, \langle \mathcal{D}, \mathcal{P}, \mathcal{E} \rangle \Longrightarrow_C \\ \langle \mathcal{D}_B \cup \{v' \mid \mathsf{v} \in \mathcal{V}_B\}, \\ \mathcal{P}_B \cup \{\mathit{cond} \Rightarrow v' = \mathcal{E}_T(\mathsf{v}) \mid \mathsf{v} \in \mathcal{V}_B\} \cup \{\neg \mathit{cond} \Rightarrow v' = \mathcal{E}_F(\mathsf{v}) \mid \mathsf{v} \in \mathcal{V}_B\}, \\ \mathcal{E}_B[\{\mathsf{v} \mapsto v' \mid \mathsf{v} \in \mathcal{V}_B\}] \rangle\end{array}}$$

where $\mathit{cond} = \mathcal{T}_F[\![\mathsf{e}, \mathcal{E}]\!]$
$\mathcal{D}_B = \mathcal{D}_T \cup \mathcal{D}_F$
$\mathcal{P}_B = \mathcal{P} \cup \{\mathit{cond} \Rightarrow f \mid f \in (\mathcal{P}_T \setminus \mathcal{P})\} \cup \{\neg \mathit{cond} \Rightarrow f \mid f \in (\mathcal{P}_F \setminus \mathcal{P})\}$
$\mathcal{E}_B = \mathcal{E}_T \cap \mathcal{E}_F$
$\mathcal{V}_B = (\mathit{domain}(\mathcal{E}_T) \cap \mathit{domain}(\mathcal{E}_F)) \setminus \mathit{domain}(\mathcal{E}_B)$

Figure 3-3: Symbolic Execution Rules. For assign, create, and branch statements, the "inline" rule ($\Longrightarrow_I$) or the "constrain" rule ($\Longrightarrow_C$) may be applied. All primed relational variables, e.g. v', are fresh.

Specification Statement Rule

A specification statement has the following form:

$$\mathsf{v}_1, \mathsf{v}_2, \ldots, \mathsf{v}_n := \textbf{spec}(\mathsf{e})$$

The symbolic execution has only a single rule for specification statements — no separate inline and constrain options. The symbolic environment that results from a specification statement maps every variable on the left-hand side of the statement to fresh relations for their value. Let $\mathcal{E}_{new}$ denote this environment:

$$\mathcal{E}_{new} = \mathcal{E}[\mathsf{v}_1 \mapsto v'_1, \mathsf{v}_2 \mapsto v'_2, \ldots, \mathsf{v}_n \mapsto v'_n]$$

The expression e on the right-hand side of the statement will be evaluated in an environment that is the same as $\mathcal{E}_{new}$, except it also maps all the old versions of the variables $\mathsf{v}_1,\ldots,\mathsf{v}_n$ to the expressions for their values in the current environment:

$$\mathcal{E}_{spec} = \mathcal{E}_{new}[\mathsf{v}_{1_{old}} \mapsto \mathcal{E}(\mathsf{v}_1), \mathsf{v}_{2_{old}} \mapsto \mathcal{E}(\mathsf{v}_2), \ldots, \mathsf{v}_{n_{old}} \mapsto \mathcal{E}(\mathsf{v}_n)]$$

The symbolic execution adds all of the fresh relations $v'_1 \ldots v'_n$ to the relational declaration; it uses $\mathcal{E}_{spec}$ to translate the expression e to a logic formula and adds the formula to the path constraint; it adds constraints that the relations $v'_1 \ldots v'_n$ contain only atoms in the types of $\mathsf{v}_1,\ldots,\mathsf{v}_n$, respectively; and it sets the symbolic environment to $\mathcal{E}_{new}$, yielding the following symbolic state after the specification statement:

$$\langle \mathcal{D} \cup \{v'_1, \ldots, v'_n\}, \mathcal{P} \cup \{\mathcal{T}_F[\![\mathsf{e}, \mathcal{E}_{spec}]\!], v'_1 \subseteq \mathcal{T}_R[\![\mathsf{T}_1, \mathcal{E}_{new}]\!], \ldots, v'_n \subseteq \mathcal{T}_R[\![\mathsf{T}_n, \mathcal{E}_{new}]\!]\}, \mathcal{E}_{new} \rangle$$

where $\mathsf{T}_1,\ldots,\mathsf{T}_n$ are the FIR types of $\mathsf{v}_1,\ldots,\mathsf{v}_n$, respectively.

Branch Statement Rules

A branch statement has the following form:

$$\textbf{if } \mathsf{e} \textbf{ then } \mathsf{S}_T \textbf{ else } \mathsf{S}_F$$

To execute a branch statement symbolically, each side of the branch is executed individually, and then the symbolic states resulting from the two sides are merged. The symbolic execution offers both an inline and a constrain rule for branch statements that differ in how they perform the merging step.

Let $\langle \mathcal{D}, \mathcal{P}, \mathcal{E} \rangle$ be the symbolic state immediately before the branch statement. Let $\langle \mathcal{D}_T, \mathcal{P}_T, \mathcal{E}_T \rangle$ be the result of symbolically executing statement S_T from state $\langle \mathcal{D}, \mathcal{P}, \mathcal{E} \rangle$ and let $\langle \mathcal{D}_F, \mathcal{P}_F, \mathcal{E}_F \rangle$ be the result of executing S_F from $\langle \mathcal{D}, \mathcal{P}, \mathcal{E} \rangle$. Both the inline and constrain strategies produce a relational declaration that contains at least the relations declared on both branches, a set we abbreviate $\mathcal{D}_B$:

$$\mathcal{D}_B = \mathcal{D}_T \cup \mathcal{D}_F$$

Both rules also share a set of path constraints. If a formula is in $\mathcal{P}_T$ but not P, then it was generated from a statement in S_T. Similarly, if a formula is in $\mathcal{P}_F$ and not P, then it was generated from S_F. Both the inline and constrain strategies yield a path constraint after the branch that contains at least the following formulas:

$$\mathcal{P}_B = \mathcal{P} \cup \{cond \Rightarrow f \mid f \in (\mathcal{P}_T \setminus \mathcal{P})\} \cup \{\neg cond \Rightarrow f \mid f \in (\mathcal{P}_F \setminus \mathcal{P})\}$$

where *cond* stands for $T_F[\![e, \mathcal{E}]\!]$, the translation of the branch condition e to a logic formula.

Both rules also yield a symbolic environment that contains at least the mappings that are common to both $\mathcal{E}_T$ and $\mathcal{E}_F$, a set of mappings we abbreviate $\mathcal{E}_B$:

$$\mathcal{E}_B = \mathcal{E}_T \cap \mathcal{E}_F$$

The inline and constrain rules differ in what additional relations they add to the symbolic state beyond $\mathcal{D}_B$, what additional formulas they add to the path constraint beyond $\mathcal{P}_B$, and to what expressions they bind variables in $\mathcal{V}_B$:

$$\mathcal{V}_B = (domain(\mathcal{E}_T) \cap domain(\mathcal{E}_F)) \setminus domain(\mathcal{E}_B)$$

These are the variables and domains bound in both $\mathcal{E}_T$ and $\mathcal{E}_F$ ("defined" on both sides of the branch in the def-use sense) which are bound to different expressions on each side of the branch.

The inline strategy does not declare any relations beyond $\mathcal{D}_B$ nor any path constraints beyond $\mathcal{P}_B$. It maps every variable or domain v in $\mathcal{V}_B$ to a conditional expression of the following form:

$$(cond \ ? \ \mathcal{E}_T(\mathsf{v}) : \mathcal{E}_F(\mathsf{v}))$$

which evaluates to the expression from the true side of the branch if the branch condition *cond* is true, and otherwise evaluates to the expression from the false side.

Thus, the inline strategy yields the following symbolic state after the branch statement:

$$\langle \mathcal{D}_B, \mathcal{P}_B, \mathcal{E}_B[\{\mathsf{v} \mapsto (cond \ ? \ \mathcal{E}_T(\mathsf{v}) : \mathcal{E}_F(\mathsf{v})) \mid \mathsf{v} \in \mathcal{V}_B\}] \rangle$$

The constrain strategy maps every variable v in $\mathcal{V}_B$ to a fresh relation v' and adds two formulas to the path constraint of the following form:

$$cond \Rightarrow v' = \mathcal{E}_T(\mathsf{v})$$
$$\neg cond \Rightarrow v' = \mathcal{E}_F(\mathsf{v})$$

So it yields the following symbolic state after the branch statement:

$$\langle \mathcal{D}_B \cup \{v' \mid \mathsf{v} \in \mathcal{V}_B\},$$
$$\mathcal{P}_B \cup \{cond \Rightarrow v' = \mathcal{E}_T(\mathsf{v}) \mid \mathsf{v} \in \mathcal{V}_B\} \cup \{\neg cond \Rightarrow v' = \mathcal{E}_F(\mathsf{v}) \mid \mathsf{v} \in \mathcal{V}_B\},$$
$$\mathcal{E}_B[\{\mathsf{v} \mapsto v' \mid \mathsf{v} \in \mathcal{V}_B\}] \rangle$$

3.2.5 Generating the Logic Problem

Once the symbolic execution of the procedure is complete, the analysis, given a specification of the procedure, constructs a relational logic problem to hand to the Kodkod model finder. The specification is provided in the form of a specification statement:

$$\mathsf{v}_1, \mathsf{v}_2, \ldots \mathsf{v}_n := \textbf{spec}(\mathsf{e})$$

The analysis first builds a symbolic environment $\mathcal{E}_{spec}$ which it will use to translate the FIR expression e to a logic formula. Let $\mathcal{E}_0$ be the initial symbolic environment and $\mathcal{E}_f$ the final environment generated by the symbolic execution. $\mathcal{E}_{spec}$ is the initial environment overridden with mappings from the variables $\mathsf{v}_1, \ldots \mathsf{v}_n$ to their expressions in the final environment and with additional mappings from the old versions of those variables to their mappings in the initial environment:

$$\begin{aligned}\mathcal{E}_{spec} = \mathcal{E}_0[\mathsf{v}_1 \mapsto \mathcal{E}_f(\mathsf{v}_1), \mathsf{v}_2 \mapsto \mathcal{E}_f(\mathsf{v}_2), \ldots, \mathsf{v}_n \mapsto \mathcal{E}_f(\mathsf{v}_n),\\ \mathsf{v}_{1old} \mapsto \mathcal{E}_0(\mathsf{v}_1), \mathsf{v}_{2old} \mapsto \mathcal{E}_0(\mathsf{v}_2), \ldots, \mathsf{v}_{nold} \mapsto \mathcal{E}_0(\mathsf{v}_n)]\end{aligned}$$

Using $\mathcal{E}_{spec}$, the expression in the specification statement is translated to a logic formula ψ and conjoined with a frame condition on the global variables not modified by the specification:

$$\psi = \mathcal{T}_F[\![\mathsf{e}, \mathcal{E}_{spec}]\!] \wedge \left(\bigwedge_{v_i \not\subseteq \{v_1 \ldots v_n\}} \mathcal{E}_f(\mathsf{v}_i) = \mathcal{E}_0(\mathsf{v}_i) \right)$$

Finally, the analysis invokes Kodkod to find an assignment to all the logic variables in the final relational declaration, D_f, such that the formulas in the following set are true:

$$\mathcal{P}_f \cup \{\neg\psi\}$$

That is, Kodkod searches for an assignment to the logic variables such that all the formulas in the final path constraint are true and the specification is false. If such a solution exists, it corresponds to a feasible trace of the procedure that violates the specification.

3.2.6 Larger Example: Web Registration

This section illustrates the symbolic execution on a larger example program. The program, found in Figure 3-4, is a modified version of the web registration program from Chapter 2. Our goal for this example is to check whether the register procedure satisfies the following specification:

$$\begin{aligned}\mathsf{newUser}, \mathsf{email}, \mathsf{id} := \textbf{spec}(\neg(\mathsf{newEmail} \subseteq \mathsf{User.email}_{old}) \Rightarrow\\ (\mathsf{newUser.email} = \mathsf{newEmail} \wedge \neg(\mathsf{newUser.id} \subseteq \mathsf{User.id}_{old})))\end{aligned}$$

```
domain User, domain String, literal Error: User
global id: User→Integer, global email: User→String
local newEmail: String, local newUser: User, local maxId: Integer local i: Integer

proc register (newEmail) : (newUser)
1  if newEmail ⊆ User.email
2     newUser := Error
   else
3     newUser := new User
4     email := email ∪ newUser→newEmail
5     maxId := spec(maxId ⊆ User.id ∧ ∀ i | i ⊆ User.id ⇒ maxId ≥ i)
6     id := id ∪ newUser→(maxId + 1)
   exit
```

Figure 3-4: Website registration procedure in FIR

which says that newUser, email, and id may change, and if the given newEmail is not in use, then in the post-state the newUser must have that email and a fresh id. Statements 5 finds the maximum id in use, and Statement 6 maps the new user to that maximum id plus 1.

The symbolic execution constructs the following initial symbolic state for the register procedure:

$$\begin{aligned}
\mathcal{D}_0 &= \{\mathit{true}, \mathit{false}, \mathit{Integer}, \mathit{Error}, \mathit{User}_0, \mathit{String}_0, \mathit{id}_0, \mathit{email}_0, \mathit{newEmail}_0\} \\
\mathcal{P}_0 &= \{\mathit{id}_0 \subseteq \mathit{User}_0 \rightarrow \mathit{Integer}, \mathit{email}_0 \subseteq \mathit{User}_0 \rightarrow \mathit{String}_0, \mathit{newEmail}_0 \subseteq \mathit{String}_0\} \\
\mathcal{E}_0 &= \{\mathsf{User} \mapsto \mathit{User}_0, \mathsf{String} \mapsto \mathit{String}_0, \mathsf{id} \mapsto \mathit{id}_0, \mathsf{email} \mapsto \mathit{email}_0, \\
&\qquad \mathsf{newEmail} \mapsto \mathit{newEmail}_0\}
\end{aligned}$$

The initial relational declaration, $\mathcal{D}_0$, contains the relations *true*, *false*, and *Integer*, for the true and false literals and Integer domain, respectively. It also contains the relation *Error* for the Error literal in the program. Lastly, it contains the relations with subscript zero for the initial values of the domains, global variables, and input parameter. The initial path constraint, $\mathcal{P}_0$, constrains each variable to be a subset (or sub-relation) of the initial value of its type, and the initial symbolic environment, $\mathcal{E}_0$, maps each domain and variable to the relation for its initial value.

Larger Example: Executing the Statements

Having constructed the initial symbolic state of the register procedure, the analysis symbolically executes the statements in its body. The analysis first encounters a branch statement, Statement 1, at which point it symbolically executes the true side of the branch, then the false side, and then merges the results. We show the symbolic execution of the body here using the inline strategy for all statements.

The first (and only) statement on the true side of the branch is Statement 2, which sets the *newUser* variable to the Error literal:

2 newUser := Error

The Error literal translates to the relation *Error* in the logic. The inline strategy binds newUser to *Error* an does not modify the relational declaration or the path constraint:

$$\begin{aligned}
\mathcal{D}_2 &= \{true, false, Integer, Error, User_0, String_0, id_0, email_0, newEmail_0\} \\
\mathcal{P}_2 &= \{id_0 \subseteq User_0 \rightarrow Integer, email_0 \subseteq User_0 \rightarrow String_0, newEmail_0 \subseteq String_0\} \\
\mathcal{E}_2 &= \{\mathsf{User} \mapsto User_0, \mathsf{String} \mapsto String_0, \mathsf{id} \mapsto id_0, \mathsf{email} \mapsto email_0, \\
&\quad \mathsf{newEmail} \mapsto newEmail_0, \ \mathsf{newUser} \mapsto Error\}
\end{aligned}$$

The difference from the initial symbolic state $\langle \mathcal{D}_0, \mathcal{P}_0, \mathcal{E}_0 \rangle$ is highlighted in gray.

The symbolic execution of the true side of the branch is complete, so the analysis now symbolically executes the false side, which begins with a create statement:

3 newUser := **new** User

The expression $\mathcal{E}_0(\mathsf{User}) = User_0$ evaluates to set of users allocated before this statement. The inline strategy binds newUser to a fresh relation, $newUser_3$, constrains $newUser_3$ to be a singleton that is not in the current set of users, and binds the User domain to be the union of the current set of users and $newUser_3$:

$$\begin{aligned}
\mathcal{D}_3 &= \{true, false, Integer, Error, User_0, String_0, id_0, email_0, newEmail_0, \\
&\quad newUser_3\} \\
\mathcal{P}_3 &= \{id_0 \subseteq User_0 \rightarrow Integer, email_0 \subseteq User_0 \rightarrow String_0, newEmail_0 \subseteq String_0, \\
&\quad \mathbf{one}\ newUser_3, \neg(newUser_3 \subseteq User_0)\} \\
\mathcal{E}_3 &= \{\mathsf{User} \mapsto (User_0 \cup newUser_3), \ \mathsf{String} \mapsto String_0, \mathsf{id} \mapsto id_0, \\
&\quad \mathsf{email} \mapsto email_0, \mathsf{newEmail} \mapsto newEmail_0, \ \mathsf{newUser} \mapsto newUser_3\}
\end{aligned}$$

The differences from the initial symbolic state are again in gray.

After Statement 3 is Statement 4, an assignment statement:

4 email := email ∪ newUser→newEmail

The symbolic execution uses the current symbolic environment, $\mathcal{E}_3$ to translate the right-hand side to the following logic expression:

$$email_0 \cup newUser_3 \rightarrow newEmail_0$$

The inline rule for assignments binds email to that expression:

$$\begin{aligned}
\mathcal{D}_4 &= \{true, false, Integer, Error, User_0, String_0, id_0, email_0, newEmail_0, \\
&\quad newUser_3\} \\
\mathcal{P}_4 &= \{id_0 \subseteq User_0 \rightarrow Integer, email_0 \subseteq User_0 \rightarrow String_0, newEmail_0 \subseteq String_0, \\
&\quad \mathbf{one}\ newUser_3, \neg(newUser_3 \subseteq User_0)\} \\
\mathcal{E}_4 &= \{\mathsf{User} \mapsto (User_0 \cup newUser_3), \mathsf{String} \mapsto String_0, \mathsf{id} \mapsto id_0, \\
&\quad \mathsf{email} \mapsto (email_0 \cup newUser_3 \rightarrow newEmail_0), \mathsf{newEmail} \mapsto newEmail_0, \\
&\quad \mathsf{newUser} \mapsto newUser_3\}
\end{aligned}$$

The differences with the state before the statement are shown in gray.

After Statement 4 is Statement 5, a specification statement:

5 maxId := **spec**(maxId $\subseteq$ User.id $\wedge$ $\forall$ i | i $\subseteq$ User.id $\Rightarrow$ maxId $\geq$ i)

The symbolic execution declares a fresh relation, $maxId_5$, for the value of maxId after the statement. Using the current environment $\mathcal{E}_4$ with an additional mapping from maxId to $maxId_5$, it translates the specification to the following logic formula:

$$maxId_5 \subseteq (User_0 \cup newUser_3).id_0 \wedge$$
$$\forall i \mid i \subseteq (User_0 \cup newUser_3).id_0 \Rightarrow sum(maxId_5) \geq sum(i)$$

Recall that the expression translation uses the *sum* function to cast set-valued expressions like $maxId_5$ to ints so that integer operators like $\geq$ may be applied.

The symbolic execution adds this formula to the path constraint, adds $maxId_5$ to the relational declaration, and binds maxId to $maxId_5$, yielding the following state:

$$\mathcal{D}_5 = \{true, false, Integer, Error, User_0, String_0, id_0, email_0, newEmail_0, newUser_3, maxId_5\}$$
$$\mathcal{P}_5 = \{id_0 \subseteq User_0 \rightarrow Integer, email_0 \subseteq User_0 \rightarrow String_0, newEmail_0 \subseteq String_0, \mathbf{one}\ newUser_3, \neg(newUser_3 \subseteq User_0), maxId_5 \subseteq (User_0 \cup newUser_3).id_0 \wedge \forall i \mid i \subseteq (User_0 \cup newUser_3).id_0 \Rightarrow sum(maxId_5) \geq sum(i)\}$$
$$\mathcal{E}_5 = \{\mathsf{User} \mapsto (User_0 \cup newUser_3), \mathsf{String} \mapsto String_0, \mathsf{id} \mapsto id_0, \mathsf{email} \mapsto (email_0 \cup newUser_3 \rightarrow newEmail_0), \mathsf{newEmail} \mapsto newEmail_0, \mathsf{newUser} \mapsto newUser_3, \mathsf{maxId} \mapsto maxId_5\}$$

The final statement of the false side of the branch is Statement 6:

6 id := id $\cup$ newUser$\rightarrow$(maxId + 1)

Using the current symbolic environment, Env_5, the right-hand sides translates to the following logic expression:

$$id_0 \cup newUser_3 \rightarrow int2expr(sum(maxId_5) + 1)$$

Recall that the expression translation uses the *int2expr* function to cast integer-valued expressions to sets before relational operators like cross product ($\rightarrow$) may be applied.

The inline strategy maps id to this expression in the symbolic environment:

$$\mathcal{D}_6 = \{true, false, Integer, Error, User_0, String_0, id_0, email_0, newEmail_0, newUser_3, maxId_5\}$$
$$\mathcal{P}_6 = \{id_0 \subseteq User_0 \rightarrow Integer, email_0 \subseteq User_0 \rightarrow String_0, newEmail_0 \subseteq String_0, \mathbf{one}\ newUser_3, \neg(newUser_3 \subseteq User_0), maxId_5 \subseteq (User_0 \cup newUser_3).id_0 \wedge \forall i \mid i \subseteq (User_0 \cup newUser_3).id_0 \Rightarrow sum(maxId_5) \geq sum(i)\}$$
$$\mathcal{E}_6 = \{\mathsf{User} \mapsto (User_0 \cup newUser_3), \mathsf{String} \mapsto String_0, \mathsf{id} \mapsto id_0 \cup newUser_3 \rightarrow int2expr(sum(maxId) + 1), \mathsf{email} \mapsto (email_0 \cup newUser_3 \rightarrow newEmail_0), \mathsf{newEmail} \mapsto newEmail_0, \mathsf{newUser} \mapsto newUser_3, \mathsf{maxId} \mapsto maxId_5\}$$

The symbolic execution of the false side of the branch is now complete. The symbolic execution will now merge the symbolic states resulting from the two sides of the branch to arrive at the final symbolic state of the procedure.

Larger Example: Merging the Sides of the Branch

Recall the branch statement that began the procedure:

```
1   if newEmail ⊆ User.email
```

Before the merge, the branch condition, newEmail $\subseteq$ User.email, is translated to a logic formula using $\mathcal{E}_0$ (the symbolic environment before the branch), yielding:

$$newEmail_0 \subseteq User_0.email_0$$

On the left below is the symbolic state $\langle \mathcal{D}_2, \mathcal{P}_2, \mathcal{E}_2 \rangle$ resulting from the true side of the branch and on the right the symbolic state $\langle \mathcal{D}_6, \mathcal{P}_6, \mathcal{E}_6 \rangle$ from the false side. Their respective differences with the initial state are shown in gray.

	$\langle \mathcal{D}_2, \mathcal{P}_2, \mathcal{E}_2 \rangle$	$\langle \mathcal{D}_6, \mathcal{P}_6, \mathcal{E}_6 \rangle$
$\mathcal{D}$	$true, false, Integer, Error$ $User_0, String_0, id_0,$ $email_0, newEmail_0$	$true, false, Integer, Error$ $User_0, String_0, id_0, email_0$ $newEmail_0,$ $newUser_3, maxId_5$
$\mathcal{P}$	$id_0 \subseteq User_0 \rightarrow Integer,$ $email_0 \subseteq User_0 \rightarrow String_0,$ $newEmail_0 \subseteq String_0$	$id_0 \subseteq User_0 \rightarrow Integer,$ $email_0 \subseteq User_0 \rightarrow String_0,$ $newEmail_0 \subseteq String_0,$ **one** $newUser_3, \neg(newUser_3 \subseteq User_0)$, $maxId_5 \subseteq (User_0 \cup newUser_3).id_0 \wedge$ $\forall i \mid i \subseteq (User_0 \cup newUser_3).id_0 \Rightarrow$ $sum(maxId_5) \geq sum(i)$
$\mathcal{E}$	User $\mapsto$ $User_0$ String $\mapsto$ $String_0$ id $\mapsto$ id_0 email $\mapsto$ $email_0$ newEmail $\mapsto$ $newEmail_0$ newUser $\mapsto$ $Error$	User $\mapsto$ $User_0 \cup newUser_3$ String $\mapsto$ $String_0$ id $\mapsto$ $id_0 \cup newUser_3 \rightarrow int2expr(sum(maxId_5)+1)$ email $\mapsto$ $email_0 \cup newUser_3 \rightarrow newEmail_0$ newEmail $\mapsto$ $newEmail_0$ newUser $\mapsto$ $newUser_3$ maxId $\mapsto$ $maxId_5$

Merging the two symbolic states yields the final symbolic state of the procedure. The final relational declaration, $\mathcal{D}_f$, is the union of the declarations from both sides:

$$\mathcal{D}_f = \{true, false, Integer, Error, User_0, String_0, id_0, email_0, newEmail_0, newUser_3, maxId_5\}$$

The final path constraint, $\mathcal{P}_f$, includes all the formulas in the initial path constraint (those not in gray on either side). No formulas were added to the path constraint by the true side of the branch (none appear in gray on the left). If there were such formulas, then for every one of those formulas f, $\mathcal{P}_f$ would have contained a formula $(newEmail_0 \subseteq User_0.email_0) \Rightarrow f$. For the three formulas added to the path constraint by the false side (those in gray on the right) $\mathcal{P}_f$ contains a formula of the form $\neg(newEmail_0 \subseteq User_0.email_0) \Rightarrow f$.

$$\begin{aligned}\mathcal{P}_f = \{&id_0 \subseteq User_0 \rightarrow Integer,\\ &email_0 \subseteq User_0 \rightarrow String_0,\\ &newEmail_0 \subseteq String_0,\\ &\neg(newEmail_0 \subseteq User_0.email_0) \Rightarrow \textbf{one}\ newUser_3,\\ &\neg(newEmail_0 \subseteq User_0.email_0) \Rightarrow \neg(newUser_3 \subseteq User_0),\\ &\neg(newEmail_0 \subseteq User_0.email_0) \Rightarrow (maxId_5 \subseteq (User_0 \cup newUser_3).id_0 \wedge\\ &\qquad \forall i \mid i \subseteq (User_0 \cup newUser_3).id_0 \Rightarrow sum(maxId_5) \geq sum(i))\}\end{aligned}$$

The final symbolic environment, $\mathcal{E}_f$, includes all the mappings for variables and domains not modified by either side. Here, those are the mappings String$\mapsto String_0$ and newEmail$\mapsto newEmail_0$. It does not contain a mapping for maxId because it is only defined (in the def-use sense) on the false side. For the remaining domains and variables — User, id, email, and newUser — $\mathcal{E}_f$ binds them to conditional expressions of the form $(newEmail_0 \subseteq User_0.email_0)\ ?\ e_T : e_F$, where e_T is its expression on the true side and e_F is its expression on the false side.

$$\begin{aligned}\mathcal{E}_f = \{&\text{String} \mapsto String_0,\\ &\text{newEmail} \mapsto newEmail_0,\\ &\text{User} \mapsto (newEmail_0 \subseteq User_0.email_0)\ ?\ User_0 : (User_0 \cup newUser_3),\\ &\text{newUser} \mapsto (newEmail_0 \subseteq User_0.email_0)\ ?\ Error : newUser_3,\\ &\text{id} \mapsto (newEmail_0 \subseteq User_0.email_0)\ ?\\ &\qquad id_0 : (id_0 \cup newUser_3 \rightarrow int2expr(sum(maxId_5)+1)),\\ &\text{email} \mapsto (newEmail_0 \subseteq User_0.email_0)\ ?\\ &\qquad email_0 : (email_0 \cup newUser_3 \rightarrow newEmail_0)\}\end{aligned}$$

The symbolic execution of the register procedure is now complete. We now show how the results of the symbolic execution are used to build the logic problem.

Larger Example: Solving the Logic Problem

Recall the specification the register procedure:

$$\begin{aligned}&\text{newUser, email, id} := \textbf{spec}(\neg(\text{newEmail} \subseteq \text{User.email}_{old}) \Rightarrow\\ &\qquad (\text{newUser.email} = \text{newEmail} \wedge \neg(\text{newUser.id} \subseteq \text{User.id}_{old})))\end{aligned}$$

The analysis constructs a symbolic environment, $\mathcal{E}_{spec}$, that is will use to translate the specification to a logic formula. $\mathcal{E}_{spec}$ is the same as the initial symbolic environment $\mathcal{E}_0$, except all of the variables in the frame condition — newUser, email, id – are bound to their expressions from the final symbolic environment, $\mathcal{E}_f$, and the old versions of email and id are bound to their expressions from $\mathcal{E}_0$:

$$\begin{aligned}\mathcal{E}_{spec} = \{&\mathsf{User} \mapsto User_0,\ \mathsf{String} \mapsto String_0,\ \mathsf{newEmail} \mapsto newEmail_0,\\ &\mathsf{email}_{old} \mapsto email_0,\ \mathsf{id}_{old} \mapsto id_0\\ &\mathsf{newUser} \mapsto (newEmail_0 \subseteq User_0.email_0)\ ?\ Error : newUser_3,\\ &\mathsf{id} \mapsto (newEmail_0 \subseteq User_0.email_0)\ ?\\ &\qquad id_0 : (id_0 \cup newUser_3 \rightarrow int2expr(sum(maxId_5)+1)),\\ &\mathsf{email} \mapsto (newEmail_0 \subseteq User_0.email_0)\ ?\\ &\qquad email_0 : (email_0 \cup newUser_3 \rightarrow newEmail_0)\}\end{aligned}$$

The analysis then uses $\mathcal{E}_{spec}$ to translate the specification to a logic formula ψ. The formula in this case is the result of replacing each of the domains and variables in the specification with its expression in $\mathcal{E}_{spec}$.

$$\begin{aligned}&\neg(newEmail_0 \subseteq User_0.email_0) \Rightarrow\\ &(((newEmail_0 \subseteq User_0.email_0)\ ?\ Error : newUser_3).\\ &\ ((newEmail_0 \subseteq User_0.email_0)\ ?\ email_0 : (email_0 \cup newUser_3 \rightarrow newEmail_0)) =\\ &\ newEmail_0 \wedge\\ &\neg(((newEmail_0 \subseteq User_0.email_0)\ ?\ Error : newUser_3).\\ &\ \ ((newEmail_0 \subseteq User_0.email_0)\ ?\ id_0 :\\ &\ \ (id_0 \cup newUser_3 \rightarrow int2expr(sum(maxId_5)+1))) \subseteq User_0.id_0))\end{aligned}$$

Forge now invokes Kodkod to find an assignment the relations in $\mathcal{D}_f$ that satisfies all the formulas in $\mathcal{P}_f$ but not ψ. In this example, if the analysis is conducted with scope and bitwidth of 3 (there are no loops so unrollings are irrelevant), Kodkod would find a solution like the following:

$$\begin{aligned}true &\mapsto \{\langle \texttt{true} \rangle\},\\ false &\mapsto \{\langle \texttt{false} \rangle\},\\ Integer &\mapsto \{\langle \texttt{-4} \rangle, \langle \texttt{-3} \rangle, \langle \texttt{-2} \rangle, \langle \texttt{-1} \rangle, \langle \texttt{0} \rangle, \langle \texttt{1} \rangle \langle \texttt{2} \rangle, \langle \texttt{3} \rangle\},\\ Error &\mapsto \{\langle \texttt{Error} \rangle\},\\ User_0 &\mapsto \{\langle \texttt{User1} \rangle, \langle \texttt{User2} \rangle\},\\ String_0 &\mapsto \{\langle \texttt{String1} \rangle, \langle \texttt{String2} \rangle, \langle \texttt{String3} \rangle\},\\ id_0 &\mapsto \{\langle \texttt{User1,3} \rangle, \langle \texttt{User2,-4} \rangle\},\\ email_0 &\mapsto \{\langle \texttt{User1,String1} \rangle, \langle \texttt{User2,String2} \rangle\},\\ newEmail_0 &\mapsto \{\langle \texttt{String3} \rangle\}\\ newUser_3 &\mapsto \{\langle \texttt{User3} \rangle\}\\ maxId_5 &\mapsto \{\langle \texttt{3} \rangle\}\end{aligned}$$

From this solution, Forge constructs a trace of the register procedure that violates the specification. Here is the procedure again, followed by the counterexample trace.

```
proc register (newEmail) : (newUser)
  if newEmail ⊆ User.email
    newUser := Error
  else
    newUser := new User
    email := email ∪ newUser→newEmail
    maxId := spec(∀ i | i ⊆ User.id ⇒ maxId ≥ i)
    id := id ∪ newUser→(maxId + 1)
  exit
```

---

initial state
- User $\mapsto$ {*User1*, *User2*}
- String $\mapsto$ {*String1*, *String2*, *String3*}
- id $\mapsto$ {⟨*User1*, *3*⟩, ⟨*User2*, *−4*⟩}
- email $\mapsto$ {⟨*User1*, *String1*⟩, ⟨*User2*, *String2*⟩}
- newEmail $\mapsto$ {*String3*}

if newEmail ⊆ User.email
- *false*

newUser := new User
- newUser $\mapsto$ {*User3*}
- User $\mapsto$ {*User1*, *User2*, *User3*}

email := email ∪ newUser→newEmail
- email $\mapsto$ {⟨*User1*, *String1*⟩, ⟨*User2*, *String2*⟩, ⟨*User3*, *String3*⟩}

maxId := spec(maxId ⊆ User.id ∧ ∀ i | i ⊆ User.id ⇒ maxId ≥ i)
- maxId $\mapsto$ {*3*}

id := id ∪ newUser→(maxId + 1)
- id $\mapsto$ {⟨*User1*, *3*⟩, ⟨*User2*, *−4*⟩, ⟨*User3*, *−4*⟩}

final state
- newUser $\mapsto$ {*User3*}
- User $\mapsto$ {*User1*, *User2*, *User3*}
- email $\mapsto$ {⟨*User1*, *String1*⟩, ⟨*User2*, *String2*⟩, ⟨*User3*, *String3*⟩}
- maxId $\mapsto$ {*3*}
- id $\mapsto$ {⟨*User1*, *3*⟩, ⟨*User2*, *−4*⟩, ⟨*User3*, *−4*⟩}

The trace begins by describing the initial state of the execution. It then lists every statement executed followed by the effect of that statement. It ends by listing the final values of all the variables and domains modified by the procedure. This trace highlights the potential for the arithmetic in the procedure to overflow the bitwidth. When a user has the maximum integer then choosing the next id to be one greater than that will cause the integer to wrap-around, possibly to an id already taken by an existing user.

In our example analysis, the bitwidth was set to 3, so the (two's-complement) integers ranged from -4 to 3. In the trace, *User1* has an id of 3, and setting the id of the new user, *User3*, to be one greater, gave it the same id as *User2*, in violation of the specification which says the new user's id must be unique. Not merely a theoretical concern, the potential for the user id's to overflow is an important issue the developer of a web registration program would need to address in real life.

3.3 A Correctness Argument

This section argues that the symbolic execution presented in this chapter is correct. It begins by describing a framework for demonstrating the correctness of a symbolic execution, a framework that is applicable more broadly than the specific symbolic execution used by the bounded verification analysis. The system could be used to prove the correctness of wide class of symbolic executions that encode the behavior of an imperative program in a logic over free variables. Next, the section applies the proof system to the symbolic execution presented in this chapter.

3.3.1 Semantics

A *program* executes over a set of variables $\mathcal{V}$. Let $PExpr^{\mathcal{V}} \supseteq \mathcal{V}$ be the set of possible program expressions over variables in $\mathcal{V}$, which must include $\mathcal{V}$ itself. Let $Stmt^{\mathcal{V}}$ be the set of possible program statements over $\mathcal{V}$.

A *program binding* over $\mathcal{V}$ is a mapping from variables in $\mathcal{V}$ to values:

$$PBind^{\mathcal{V}} \equiv \mathcal{V} \rightarrow Value$$

For a program binding pb and a variable v, $pb(v)$ denotes the value to which v is mapped in pb. More generally, we lift the $pb(\cdot)$ function over any expression $pe \in PExpr^{\mathcal{V}}$, so that $pb(pe)$ denotes the evaluation of that expression in the binding pb, as defined by the semantics of the programming language. (We assume the semantics of evaluating a variable in a binding is always the result of looking up the value of that variable in the binding, so $pb(v)$ is unambiguous.)

For a program $\mathcal{V}$, $\mathcal{B}_0^{\mathcal{V}}$ is the set of possible bindings at the start of the program:

$$\mathcal{B}_0^{\mathcal{V}} \subseteq 2^{PBind^{\mathcal{V}}}$$

The $Exec^{\mathcal{V}}$ function gives the semantics of program execution. It accepts a statement and a binding and yields the set of bindings that may result from that statement:

$$Exec^{\mathcal{V}}\colon\ Stmt^{\mathcal{V}} \rightarrow PBind^{\mathcal{V}} \rightarrow 2^{PBind^{\mathcal{V}}}$$

The *Exec* function yields a *set* of bindings, as opposed to exactly one, to account for possible non-determinism in the program (e.g. as allowed by specification statements).

A *declaration* consists of a set of logic variables $\mathcal{D}$. Let $LExpr^{\mathcal{D}} \supseteq \mathcal{D}$ and $Form^{\mathcal{D}}$ be the set of expressions and formulas, respectively, that can be constructed from variables in $\mathcal{D}$. A *logic binding* over $\mathcal{D}$, is a mapping from variables in $\mathcal{D}$ to values:

$$LBind^{\mathcal{D}} \equiv \mathcal{D} \rightarrow Value$$

For a logic binding lb and a logic variable d, $lb(d)$ denotes the value to which d is mapped in lb. We lift the function $lb(\cdot)$ over any expression $le \in LExpr^{\mathcal{D}}$, so that $lb(le)$ denotes the evaluation of that expression to a value in lb, as defined by the semantics of the logic. We also overload $lb(\cdot)$ so that for any formula $f \in Form^{\mathcal{D}}$, $lb(f)$ denotes the Boolean predicate that is true if and only if f evaluates to true in lb. Let $solutions^{\mathcal{D}}(f)$ be the set of all logic bindings over $\mathcal{D}$ in which f is true:

$$solutions^{\mathcal{D}}(f) \equiv \{lb \in LBind^{\mathcal{D}} \mid lb(f)\}$$

3.3.2 Symbolic Execution

For a program $\mathcal{V}$ and declaration $\mathcal{D}$, a *symbolic environment* $Env^{\mathcal{V},\mathcal{D}}$ is a mapping of variables in $\mathcal{V}$ to logic expressions over $\mathcal{D}$:

$$Env^{\mathcal{V},\mathcal{D}} \equiv \mathcal{V} \rightarrow LExpr^{\mathcal{D}}$$

The state of a symbolic execution of a program $\mathcal{V}$ is a triple:

$$State^{\mathcal{V}} \equiv \langle \mathcal{D}, \mathcal{P}, \mathcal{E} \rangle$$

consisting of the following:

$\mathcal{D}$: a set of logic variables
$\mathcal{P} \in Form^{\mathcal{D}}$: a logic formula over $\mathcal{D}$
$\mathcal{E} \in Env^{\mathcal{V},\mathcal{D}}$: a symbolic environment that maps program variables in $\mathcal{V}$ to logic expressions over $\mathcal{D}$

For a program $\mathcal{V}$, a *symbolic execution* consists of an initial symbolic state:

$$\langle \mathcal{D}_0, \mathcal{P}_0, \mathcal{E}_0 \rangle \in State^{\mathcal{V}}$$

a symbolic execution function, $Sym^{\mathcal{V}}$, which, given a statement and symbolic state over $\mathcal{V}$, yields a new symbolic state:

$$Sym^{\mathcal{V}}\colon\ Stmt^{\mathcal{V}} \rightarrow State^{\mathcal{V}} \rightarrow State^{\mathcal{V}}$$

and an expression translation function $\mathcal{T}^{\mathcal{V},\mathcal{D}}$ for translating program expressions in a symbolic environment to logic expressions:

$$\mathcal{T}^{\mathcal{V},\mathcal{D}}\colon\ PExpr^{\mathcal{V}} \rightarrow Env^{\mathcal{V},\mathcal{D}} \rightarrow LExpr^{\mathcal{D}}$$

3.3.3 Proof Framework

Given a symbolic environment $\mathcal{E} \in Env^{\mathcal{V},\mathcal{D}}$, a logic binding $lb \in LBind^{\mathcal{D}}$ *encodes* a program binding $pb \in PBind^{\mathcal{V}}$ with respect to $\mathcal{E}$ if, for every variable in $\mathcal{V}$, looking up the value of that variable in pb yields the same value as looking up the logic expression for that variable in $\mathcal{E}$ and evaluating that logic expression in lb:

$$encodes^{\mathcal{V},\mathcal{D}}(lb, pb, \mathcal{E}) \equiv \forall\ v \in \mathcal{V} \mid pb(v) = lb(\mathcal{E}(v))$$

For a set of program bindings $\mathcal{B} \subseteq 2^{PBind^{\mathcal{V}}}$, a symbolic state $\langle \mathcal{D}, \mathcal{P}, \mathcal{E} \rangle$ is a *sound encoding* of $\mathcal{B}$ if every solution to $\mathcal{P}$ encodes a binding in $\mathcal{B}$:

$$sound(\mathcal{D}, \mathcal{P}, \mathcal{E}, \mathcal{B}) \equiv \forall\, lb \in solutions^{\mathcal{D}}(\mathcal{P}) \mid \exists\, pb \in \mathcal{B} \mid encodes(lb, pb, \mathcal{E})$$

A symbolic state $\langle \mathcal{D}, \mathcal{P}, \mathcal{E} \rangle$ is a *complete encoding* if every binding in $\mathcal{B}$ is encoded by some solution to $\mathcal{P}$:

$$complete(\mathcal{D}, \mathcal{P}, \mathcal{E}, \mathcal{B}) \equiv \forall\, pb \in \mathcal{B} \mid \exists\, lb \in solutions^{\mathcal{D}}(\mathcal{P}) \mid encodes(lb, pb, \mathcal{E})$$

A symbolic state is a *correct encoding* of $\mathcal{B}$ if it is both sound and complete.

With these definitions, we can now prove by induction that a symbolic execution is correct by showing that the symbolic state that it generates at each program point is a correct encoding of the reachable program bindings at that point: that the state is sound (i.e., it includes *only* states the program can reach) and complete (i.e., it includes *all* the states the program can reach).

Proof by Induction

For a program $\mathcal{V}$, the base case must demonstrate that $\mathcal{D}_0, \mathcal{P}_0, \mathcal{E}_0$, the initial symbolic state produced by the symbolic execution, correctly encodes $\mathcal{B}_0$, the set of initial valid bindings to the program, i.e., it must demonstrate $sound(\mathcal{D}_0, \mathcal{P}_0, \mathcal{E}_0, \mathcal{B}_0)$ and $complete(\mathcal{D}_0, \mathcal{P}_0, \mathcal{E}_0, \mathcal{B}_0)$.

The inductive case assumes that a current symbolic state $\langle \mathcal{D}, \mathcal{P}, \mathcal{E} \rangle$ is a correct encoding of a set of program bindings $\mathcal{B}$ for a program $\mathcal{V}$. For any statement $s \in Stmt^{\mathcal{V}}$, let $\mathcal{B}'$ be the set of feasible program bindings after the statement:

$$\mathcal{B}' = \{pb' \in PBind^{\mathcal{V}} \mid \exists\, pb : \mathcal{B} \mid pb' \in Exec(s, pb)\}$$

For the same statement s, let $\langle \mathcal{D}', \mathcal{P}', \mathcal{E}' \rangle$ be the symbolic state after the statement:

$$\langle \mathcal{D}', \mathcal{P}', \mathcal{E}' \rangle = Sym(s, \langle \mathcal{D}, \mathcal{P}, \mathcal{E} \rangle)$$

The inductive case must demonstrate that $\langle \mathcal{D}', \mathcal{P}', \mathcal{E}' \rangle$ is a correct encoding of $\mathcal{B}'$ — $sound(\mathcal{D}', \mathcal{P}', \mathcal{E}', \mathcal{B}')$ and $complete(\mathcal{D}', \mathcal{P}', \mathcal{E}', \mathcal{B}')$ — for every statement $s \in Stmt^{\mathcal{V}}$.

Correctness of Expression Translation

Since statements in the program contain expressions, demonstrating the inductive case will rely upon a correct translation from program expressions to logic expressions. To prove the correctness of the expression translation function $\mathcal{T}^{\mathcal{V},\mathcal{D}}$ one must demonstrate the following. Consider any program binding pb and any logic binding lb that that encodes it with respect to a symbolic environment $\mathcal{E} \in Env^{\mathcal{V},\mathcal{D}}$. Given any such pb, lb, and $\mathcal{E}$, evaluating any program expression $pe \in PExpr^{\mathcal{V}}$ in the program binding pb should yield the same value as translating pe to a logic expression and evaluating that logic expression in the logic binding lb:

$$\begin{aligned} &\forall \mathcal{E} \in Env^{\mathcal{V},\mathcal{D}}, pb \in PBind^{\mathcal{V}}, lb \in LBind^{\mathcal{D}} \mid encodes^{\mathcal{V},\mathcal{D}}(lb, pb, \mathcal{E}) \Rightarrow \\ &\quad \forall\, pe \in PExpr^{\mathcal{V}} \mid pb(pe) = lb(\mathcal{T}^{\mathcal{V},\mathcal{D}}(pe, \mathcal{E})) \end{aligned}$$

3.3.4 Applying the Proof System to Forge

We now apply the proof system to the Forge symbolic execution rules given in this chapter. When applied to the our symbolic execution, the set of variables $\mathcal{V}$ to which the proof refers are the variables and user-defined domains declared by the FIR program. The set of program expressions $PExpr^{\mathcal{V}}$ and statements $Stmt^{\mathcal{V}}$ are those that can be constructed according to the FIR grammar (Figure 2.1 of Chapter 2). The variables in the logic declaration $\mathcal{D}$ become relations in the Kodkod relational logic, and the expressions in $LExpr^{\mathcal{D}}$ and formulas in $Form^{\mathcal{D}}$ are those that can be constructed following the grammar of the relational logic (Figure 3-1 of this chapter). The set of *Values* to which bindings map program and logic variables are relational *constants*, i.e., fixed sets of tuples.

What follows is not intended to be a rigorous proof, but a sufficiently convincing sketch that persuades the reader of the correctness of our symbolic execution and provides enough guidance so that readers could complete the proof on their own. Below we argue for the correctness of the initial state of the symbolic execution and for the correctness of the inline symbolic execution rules for assignment and branch statements.

Correctness of Initial State

From a FIR program with a set of variables and user-defined domains v_1, v_2, ..., v_n, recall that the symbolic execution constructs an initial symbolic environment $\mathcal{E}_0$ that maps each v_i to a fresh relation v_{i_0}. We first prove that a logic binding lb_0 encodes a program binding pb_0, both of the following form, with respect to $\mathcal{E}_0$:

$$lb_0 = \{v_{1_0} \mapsto c_1, v_{2_0} \mapsto c_2, \ldots, v_{n_0} \mapsto c_n\}$$
$$pb_0 = \{\mathsf{v}_1 \mapsto c_1, \mathsf{v}_2 \mapsto c_2, \ldots, \mathsf{v}_n \mapsto c_n\}$$

The logic binding lb_0 encodes the program binding pb_0 if, for all program variables v_i, the following holds:

$$pb_0(\mathsf{v}_i) = lb_0(\mathcal{E}_0(\mathsf{v}_i))$$

Since $pb_0(\mathsf{v}_i) = c_i$, $\mathcal{E}_0(\mathsf{v}_i) = v_{i_0}$, and $lb_0(v_{i_0}) = c_i$, both sides are equal to c_i, so lb_0 encodes pb_0.

Proving that $\langle \mathcal{D}_0, \mathcal{P}_0, \mathcal{E}_0 \rangle$ is a correct encoding requires showing that a logic binding lb_0 of the form above is in $solutions^{\mathcal{D}_0}(\mathcal{P}_0)$ if and only if the program binding pb_0 of the form above in an initial program binding. (That lb_0 encodes pb_0 was just demonstrated.) This is true by construction. The feasible initial program states to a FIR program are ones in which the domains, global variables, and input parameters to the procedure are bound to constants of the same arity, the only restriction being that the global variables and input parameters are subsets (or sub-relations) of the initial value of their types. Those program states correspond exactly to the solutions permitted by the initial path constraint $\mathcal{P}_0$.

Inline Rule for Assignments. A FIR assignment statement has the form $\mathsf{x} := \mathsf{e}$. Executing an assignment from a program binding pb deterministically yields a single program binding pb' in which x is bound to the evaluation of e in pb and in which the bindings for all other variables have stayed the same, i.e., $pb' = pb[\mathsf{x} \mapsto pb(\mathsf{e})]$. Thus, $\mathcal{B}'$ after an assignment statement is as follows:

$$\mathcal{B}' = \{pb' \mid \exists\, pb \in \mathcal{B} \mid pb' = pb[\mathsf{x} \mapsto pb(\mathsf{e})]\}$$

Recall that the inline rule for the assignment maps x to the translation of the expression e and does not alter the relational declaration or path constraint:

$$\langle \mathcal{D}', \mathcal{P}', \mathcal{E}' \rangle = \langle \mathcal{D}, \mathcal{P}, \mathcal{E}[\mathsf{x} \mapsto \mathcal{T}(\mathsf{e}, \mathcal{E})] \rangle$$

To show this is correct, we first show that it is a sound encoding:

$$\forall\, lb' \in solutions^{\mathcal{D}'}(\mathcal{P}') \mid \exists\, pb' \in \mathcal{B}' \mid encodes(lb', pb', \mathcal{E}')$$

which, given $\langle \mathcal{D}', \mathcal{P}', \mathcal{E}' \rangle = \langle \mathcal{D}, \mathcal{P}, \mathcal{E}[\mathsf{x} \mapsto \mathcal{T}(\mathsf{e}, \mathcal{E})] \rangle$, reduces to:

$$\forall\, lb \in solutions^{\mathcal{D}}(\mathcal{P}) \mid \exists\, pb' \in \mathcal{B}' \mid encodes(lb, pb', \mathcal{E}[\mathsf{x} \mapsto \mathcal{T}(\mathsf{e}, \mathcal{E})])$$

Consider any logic binding $lb \in solutions^{\mathcal{D}}(\mathcal{P})$. By the assumption that $\langle \mathcal{D}, \mathcal{P}, \mathcal{E} \rangle$ soundly encodes $\mathcal{B}$, there must be a $pb \in \mathcal{B}$ such that $encodes(lb, pb, \mathcal{E})$. Let $pb' = pb[\mathsf{x} \mapsto pb(\mathsf{e})]$, which must be a member of $\mathcal{B}'$ by the definition of $\mathcal{B}'$ above. It remains to show that lb encodes pb' with respect to $\mathcal{E}'$:

$$\forall\, \mathsf{v} \in \mathcal{V} \mid pb'(\mathsf{v}) = lb(\mathcal{E}'(\mathsf{v}))$$

Since lb encodes pb with respect to $\mathcal{E}$, and since pb' differs from pb and $\mathcal{E}'$ from $\mathcal{E}$ only in x, it suffices to show that $pb'(\mathsf{x}) = lb(\mathcal{E}'(\mathsf{x}))$. By the definition of pb' and $\mathcal{E}'$, this reduces to $pb(\mathsf{e}) = lb(\mathcal{T}(\mathsf{e}, \mathcal{E}))$, which is true by the assumed correctness of the expression translation.

Given $\langle \mathcal{D}', \mathcal{P}', \mathcal{E}' \rangle = \langle \mathcal{D}, \mathcal{P}, \mathcal{E}[\mathsf{x} \mapsto \mathcal{T}(\mathsf{e}, \mathcal{E})] \rangle$, proving completeness of the encoding reduces to:

$$\forall\, pb' \in \mathcal{B}' \mid \exists\, lb \in solutions^{\mathcal{D}}(\mathcal{P}) \mid encodes(lb, pb', \mathcal{E}[\mathsf{x} \mapsto \mathcal{T}(\mathsf{e}, \mathcal{E})])$$

Consider any program binding $pb' \in \mathcal{B}'$. By the definition of $\mathcal{B}'$, there must exist some $pb \in \mathcal{B}$ such that $pb' = pb[\mathsf{x} \mapsto pb(\mathsf{e})]$. By the assumption that $\langle \mathcal{D}, \mathcal{P}, \mathcal{E} \rangle$ completely encodes $\mathcal{B}$, there must exist a logic binding $lb \in solutions^{\mathcal{D}}(\mathcal{P})$ such that $encodes(lb, pb, \mathcal{E})$. It remains to show that lb encodes pb' with respect to $\mathcal{E}'$. Since lb encodes pb with respect to $\mathcal{E}$, and since pb' differs from pb and $\mathcal{E}'$ from $\mathcal{E}$ only in x, it suffices to show that $pb'(\mathsf{x}) = lb(\mathcal{E}'(\mathsf{x}))$. By the definition of pb' and $\mathcal{E}'$, this reduces to $pb(\mathsf{e}) = lb(\mathcal{T}(\mathsf{e}, \mathcal{E}))$, which is true by the assumed correctness of the expression translation.

Solution Monotonicity Proving the correctness of some of the symbolic execution rules, including the inline rule for branch statements below, relies on the *solution monotonicity* of our symbolic execution, defined as follows. Consider any logic binding $lb' \in solutions^{\mathcal{D}'}(\mathcal{P}')$, where $\langle \mathcal{D}', \mathcal{P}', \mathcal{E}' \rangle = Sym(s, \langle \mathcal{D}, \mathcal{P}, \mathcal{E} \rangle)$ for any $s \in Stmt^{\mathcal{V}}$. Let $\mathcal{D} \triangleleft lb'$ denote the logic binding formed by restricting lb' to the logic variables in $\mathcal{D}$. The *solution monotonicity* property says that $(\mathcal{D} \triangleleft lb') \in solutions^{\mathcal{D}}(\mathcal{P})$. This is true because the symbolic execution rules only adds, never removes, logic variables and path constraints to the symbolic state.

Inline Rule for Branch Statements. We assume again that a current symbolic state $\langle \mathcal{D}, \mathcal{P}, \mathcal{E} \rangle$ correctly encodes set of program bindings $\mathcal{B}$. For a FIR branch statement of the form **if** e **then** S_T **else** S_F, let $\langle \mathcal{D}_T, \mathcal{P}_T, \mathcal{E}_T \rangle = Sym(\mathsf{S}_T, \langle \mathcal{D}, \mathcal{P}, \mathcal{E} \rangle)$ and $\langle \mathcal{D}_F, \mathcal{P}_F, \mathcal{E}_F \rangle = Sym(\mathsf{S}_F, \langle \mathcal{D}, \mathcal{P}, \mathcal{E} \rangle)$. Using induction on the structure of the branch statement, this proof also assumes that $\langle \mathcal{D}_T, \mathcal{P}_T, \mathcal{E}_T \rangle$ correctly encodes $\mathcal{B}_T$ and $\langle \mathcal{D}_F, \mathcal{P}_F, \mathcal{E}_F \rangle$ correctly encodes $\mathcal{B}_F$, where $\mathcal{B}_T$ and $\mathcal{B}_F$ are defined as follows:

$$\mathcal{B}_T = \{pb_T \mid \exists\, pb \in \mathcal{B} \mid pb_T \in \mathit{Exec}(\mathsf{S}_T, pb)\}$$
$$\mathcal{B}_F = \{pb_F \mid \exists\, pb \in \mathcal{B} \mid pb_F \in \mathit{Exec}(\mathsf{S}_F, pb)\}$$

Executing the branch statement from a set of possible program bindings $\mathcal{B}$ will yield the set of program bindings $\mathcal{B}'$, defined as follows:

$$\mathcal{B}' = \{pb' \mid \exists\, pb \in \mathcal{B} \mid (pb(\mathsf{e}) \wedge pb' \in \mathit{Exec}(\mathsf{S}_T, pb)) \vee (\neg pb(\mathsf{e}) \wedge pb' \in \mathit{Exec}(\mathsf{S}_F, pb))\}$$

Recall that the inline rule for branch statements yields the following symbolic state:

$$\langle \mathcal{D}', \mathcal{P}', \mathcal{E}' \rangle = \langle \mathcal{D}_T \cup \mathcal{D}_F, \mathcal{P} \cup \{cond \Rightarrow f \mid f \in \mathcal{P}_T \setminus \mathcal{P}\} \cup \{\neg cond \Rightarrow f \mid f \in \mathcal{P}_F \setminus \mathcal{P}\}, \mathcal{E}[\{\mathsf{v} \mapsto (cond\ ?\ \mathcal{E}_T(\mathsf{v}) : \mathcal{E}_F(\mathsf{v})) \mid \mathsf{v} \in \mathcal{V}\}]\rangle$$

where $cond$ stands for $\mathcal{T}(\mathsf{e}, \mathcal{E})$.

Soundness of Encoding. To show that $\langle \mathcal{D}', \mathcal{P}', \mathcal{E}' \rangle$ is a sound encoding of $\mathcal{B}'$, consider any $lb' \in \mathit{solutions}^{\mathcal{D}'}(\mathcal{P}')$. Let $lb = \mathcal{D} \triangleleft lb'$. By the solution monotonicity property, $lb \in \mathit{solutions}^{\mathcal{D}}(\mathcal{P})$. By the assumption that $\langle \mathcal{D}, \mathcal{P}, \mathcal{E} \rangle$ is a sound encoding of $\mathcal{B}$, there must exist a $pb \in \mathcal{B}$ that is encoded by lb. Since lb encodes pb, we know from the correctness of the expression translation that $pb(\mathsf{e}) = lb(T(\mathsf{e}, \mathcal{E})) = lb(cond)$. We now split the proof into two cases: when $pb(\mathsf{e}) = lb(cond) = true$ and when $pb(\mathsf{e}) = lb(cond) = false$.

When $pb(\mathsf{e}) = lb(cond) = true$, $\mathcal{B}'$ and $\langle \mathcal{D}', \mathcal{P}', \mathcal{E}' \rangle$ simplify to the following:

$$\mathcal{B}' = \mathcal{B}_T = \{pb' \mid \exists\, pb \in \mathcal{B} \mid pb' \in \mathit{Exec}(\mathsf{S}_T, pb)\}$$
$$\langle \mathcal{D}', \mathcal{P}', \mathcal{E}' \rangle = \langle \mathcal{D}_T \cup \mathcal{D}_F, \mathcal{P}_T, \mathcal{E}_T \rangle$$

It remains to show that $\langle \mathcal{D}_T \cup \mathcal{D}_F, \mathcal{P}_T, \mathcal{E}_T \rangle$ is a sound encoding of $\mathcal{B}_T$. From any $lb' \in \mathit{solutions}^{\mathcal{D}_T \cup \mathcal{D}_F}(\mathcal{P}_T)$, construct the logic binding $lb_T = (\mathcal{D}_T \triangleleft lb')$. Since $\mathcal{P}_T$ does not involve any logic variables in $(\mathcal{D}_F \setminus \mathcal{D}_T)$, it follows that $lb_T \in \mathit{solutions}^{\mathcal{D}_T}(\mathcal{P}_T)$. From the assumption that $\langle \mathcal{D}_T, \mathcal{P}_T, \mathcal{E}_T \rangle$ is a sound encoding of $\mathcal{B}_T$, there must exist $pb' \in \mathcal{B}_T$ that is encoded by lb_T. Since $\mathcal{E}_T$ doesn't involve any logic variables in $(\mathcal{D}_F \setminus \mathcal{D}_T)$, it follows that lb' encodes pb' as well.

The case when $pb(\mathsf{e}) = lb(cond) = false$ can be demonstrated analogously.

Completeness of Encoding. To show that $\langle \mathcal{D}', \mathcal{P}', \mathcal{E}' \rangle$ is a complete encoding of $\mathcal{B}'$, consider any $pb' \in \mathcal{B}'$. By definition of $\mathcal{B}'$, there exists $pb \in \mathcal{B}$ such that $(pb(\mathsf{e}) \wedge pb' \in \mathit{Exec}(\mathsf{S}_T, pb)) \vee (\neg pb(\mathsf{e}) \wedge pb' \in \mathit{Exec}(\mathsf{S}_F, pb))$. By our assumption that $\mathcal{B}$ is completely encoded by $\langle \mathcal{D}, \mathcal{P}, \mathcal{E} \rangle$, there must exist $lb \in \mathit{solutions}^{\mathcal{D}}(\mathcal{P})$ that encodes pb. Since lb encodes pb, we know from the correctness of the expression translation that $pb(\mathsf{e}) = lb(T(\mathsf{e}, \mathcal{E})) = lb(cond)$. We now split the proof into two cases: when $pb(\mathsf{e}) = lb(cond) = true$ and when $pb(\mathsf{e}) = lb(cond) = false$.

When $pb(\mathsf{e}) = lb(cond) = true$, $\mathcal{B}'$ and $\langle \mathcal{D}', \mathcal{P}', \mathcal{E}' \rangle$ again simplify to the following:

$$\mathcal{B}' = \mathcal{B}_T = \{pb' \mid \exists\, pb \in \mathcal{B} \mid pb' \in \mathit{Exec}(\mathsf{S}_T, pb)\}$$
$$\langle \mathcal{D}', \mathcal{P}', \mathcal{E}' \rangle = \langle \mathcal{D}_T \cup \mathcal{D}_F, \mathcal{P}_T, \mathcal{E}_T \rangle$$

It remains to show that $\langle \mathcal{D}_T \cup \mathcal{D}_F, \mathcal{P}_T, \mathcal{E}_T \rangle$ is a complete encoding of $\mathcal{B}_T$. For any $pb' \in \mathcal{B}_T$, by our assumption that $\langle \mathcal{D}_T, \mathcal{P}_T, \mathcal{E}_T \rangle$ is a complete encoding of $\mathcal{B}_T$, there must exist $lb_T \in solutions^{\mathcal{D}_T}(\mathcal{P}_T)$ that encodes pb'. Construct a logic binding lb' such that $(\mathcal{D}_T \triangleleft lb') = lb_T$ and where the variables in $\mathcal{D}_F \setminus \mathcal{D}_T$ are assigned arbitrary values. Since $\mathcal{P}_T$ does not involve any logic variables in $(\mathcal{D}_F \setminus \mathcal{D}_T)$, it follows that $lb' \in solutions^{\mathcal{D}_T \cup \mathcal{D}_F}(\mathcal{P}_T)$. Since $\mathcal{E}_T$ doesn't involve any logic variables in $(\mathcal{D}_F \setminus \mathcal{D}_T)$, it follows that lb' encodes pb' as well.

The case when $pb(\mathsf{e}) = lb(cond) = false$ can be demonstrated analogously.

Correctness of FIR Expression Translation

Recall from above that to prove the correctness of the expression translation function, $\mathcal{T}$, one must show that for every program binding pb and logic binding lb, where lb encodes pb with respect to an symbolic environment $\mathcal{E}$, the following holds:

$$\forall\ pe \in PExpr^{\mathcal{V}} \mid pb(pe) = lb(\mathcal{T}(pe, \mathcal{E}))$$

This can be proved by induction on the structure of program expressions $pe \in PExpr^{\mathcal{V}}$. The following sections give the base case and an example inductive case of that proof.

Base Case of Expression Translation. The base case of that induction shows that $\mathcal{T}$ correctly translates all program expressions that are variables:

$$\forall\ \mathsf{v} \in \mathcal{V} \mid pb(\mathsf{v}) = lb(\mathcal{T}(\mathsf{v}, \mathcal{E}))$$

The translation of a variable v in an environment $\mathcal{E}$, $\mathcal{T}(v, \mathcal{E})$, is defined (Figure 3-2) to be the result of looking up the expression to which that variable is bound in the environment, $\mathcal{E}(v)$. So the base case reduces to demonstrating the following:

$$\forall\ \mathsf{v} \in \mathcal{V} \mid pb(\mathsf{v}) = lb(\mathcal{E}(\mathsf{v}))$$

which is exactly the definition of consistency of pb and lb that we are assuming.

Inductive Case of Expression Translation. The inductive must be demonstrated for every kind of FIR composite expression. All of these are fairly trivial examples that follow straight from the semantics of FIR expressions. We show here an example induction for expressions formed with the FIR union ($\cup$) operator. The inductive case assumes the existence of two FIR expressions, x and y, which for any pair of bindings, lb and pb, where lb encodes pb, are translated correctly:

$$pb(\mathsf{x}) = lb(\mathcal{T}(\mathsf{x}, \mathcal{E}))$$
$$pb(\mathsf{y}) = lb(\mathcal{T}(\mathsf{y}, \mathcal{E}))$$

With this assumption, we will show that the union expression $(\mathsf{x} \cup_P \mathsf{y})$ is translated correctly:

$$pb(\mathsf{x} \cup_P \mathsf{y}) = lb(\mathcal{T}(\mathsf{x} \cup_P \mathsf{y}, \mathcal{E}))$$

We use the subscripts to distinguish the union operator in FIR ($\cup_P$), from a union in the logic ($\cup_L$), from a union of constants ($\cup_C$). The semantics of evaluating a union in FIR is the union of the evaluation of two operands, so we can rewrite the left hand side as follows:

$$pb(\mathsf{x}) \cup_C pb(\mathsf{y}) = lb(\mathcal{T}(\mathsf{x} \cup_P \mathsf{y}, \mathcal{E}))$$

The expression translation for a FIR union expression (as given in Figure 3-2) performs a union of the translation of the subexpressions, which we can use to rewrite the right-hand side:

$$pb(\mathsf{x}) \cup_C pb(\mathsf{y}) = lb(\mathcal{T}(\mathsf{x}, \mathcal{E}) \cup_L \mathcal{T}(\mathsf{y}, \mathcal{E}))$$

And the semantics of evaluating a union in relational logic is the union of the evaluation of two operands:

$$pb(\mathsf{x}) \cup_C pb(\mathsf{y}) = lb(\mathcal{T}(\mathsf{x}, \mathcal{E})) \cup_C lb(\mathcal{T}(\mathsf{y}, \mathcal{E}))$$

This formula follows directly from the assumptions of the inductive case that $pb(\mathsf{x}) = lb(\mathcal{T}(\mathsf{x}, \mathcal{E}))$ and $pb(\mathsf{y}) = lb(\mathcal{T}(\mathsf{y}, \mathcal{E}))$.

3.3.5 Summary of Correctness Argument

This section provided a framework for proving the correctness of a symbolic execution and applied it to parts of the Forge symbolic execution to produce the following:

- A partial structural induction proof that the translation from FIR expressions to logic expressions is correct, including a base case involving the translation of FIR variables, and an inductive case involving the translation of union expressions.
- A partial proof by induction that the symbolic execution is correct, including a base case that the initial symbolic state is correct, and an inductive case that the inline symbolic execution rules for assign and branch statements are correct.

The goal here was not a completely rigorous proof, but rather enough of an argument to convey the basic sense in which the Forge symbolic execution is correct and to provide intuition for how the rest of the symbolic execution could be proven correct in a similar manner.

3.4 Breaking Symmetries on Dynamic Allocation

This chapter presents an optimization to the symbolic execution that improves the performance of the bounded verification analysis. The optimized symbolic execution presented here generates relational logic problems that the Kodkod model finder can solve in less time, compared to the original symbolic execution presented in Section 3.2. The optimization exploits the symmetries in the order of allocation of atoms from a domain.

3.4.1 Isomorphic Allocation Orders

Consider, for example, a simple procedure that creates two marbles:

```
domain Marble

proc makeTwo () : ()
1  a := new Marble
2  b := new Marble
```

To analyze this procedure, the symbolic execution previously described, regardless of the strategy, would declare at least three relations: a relation $Marble_0$ for the initial value of the Marble domain, and relations a_1 and b_2 for the values assigned to the a and b locals. In a scope of three marble atoms: $\{\langle \texttt{M1} \rangle, \langle \texttt{M2} \rangle, \langle \texttt{M3} \rangle\}$, there are 12 feasible traces of this code — a and b can be assigned according to any permutation of 2 atoms chosen from the 3, and the initial value of Marble either contains the remaining atom or it does not:

Marble_{old}	a	b	Marble	Marble_{old}	a	b	Marble
$Marble_0$	a_1	b_2	$Marble_0 \cup a_1 \cup b_2$	$Marble_0$	a_1	b_2	$Marble_0 \cup a_1 \cup b_2$
{}	M1	M2	{M1, M2}	{M3}	M1	M2	{M1, M2, M3}
{}	M2	M1	{M1, M2}	{M3}	M2	M1	{M1, M2, M3}
{}	M1	M3	{M1, M3}	{M2}	M1	M3	{M1, M2, M3}
{}	M3	M1	{M1, M3}	{M2}	M3	M1	{M1, M2, M3}
{}	M2	M3	{M2, M3}	{M1}	M2	M3	{M1, M2, M3}
{}	M3	M2	{M2, M3}	{M1}	M3	M2	{M1, M2, M3}

The key observation exploited by the optimization is that all of the traces on the left-hand side of the table are *isomorphic* to one another, as are all those on the right. Conceptually, there are only two potential traces to the procedure — one in which Marble_{old} is empty and one in which it is a singleton — but these two conceptual traces are expanded to the 12 above by permuting the atom names. For example, swapping M1 and M2 in the top execution on either side yields the execution second from the top. Given the execution second from the top, replacing M1 with M3, M3 with M2, and M2 with M1 yields the execution third from the top. Each side is said to form an *equivalence class* of isomorphic executions.

Since a specification cannot reference the name of an atom, an execution satisfies a specification if and only if every execution in its equivalence class satisfies it. As a result, it would be wasteful for Forge to check all 12 executions above against the specification; it would suffice instead to check one execution from each side of the table. This process is known as *symmetry breaking*. Symmetry breaking is the elimination of solutions (in our setting, executions), that leaves at least one *representative* solution from each equivalence class.

Without any hints from our analysis, Kodkod performs some symmetry breaking on our marble example automatically. Specifically, it imposes a lex-leader symmetry breaking predicate [77] on an arbitrarily-chosen relation. The lex-leader predicate

constrains the chosen relation to contain atoms that belong to a prefix of the lexical ordering of the atoms. In our example, it would say that if the relation contains `M3`, then it must contain `M2`, and if it contains `M2` it must contain `M1`. If this predicate were imposed on $Marble_0$, then it would reduce the 12 possible executions to 8:

Marble_{old}	a	b	Marble	Marble_{old}	a	b	Marble
$Marble_0$	a_1	b_2	$Marble_0 \cup a_1 \cup b_2$	$Marble_0$	a_1	b_2	$Marble_0 \cup a_1 \cup b_2$
{}	M1	M2	{M1, M2}	{M1}	M2	M3	{M1, M2, M3}
{}	M2	M1	{M1, M2}	{M1}	M3	M2	{M1, M2, M3}
{}	M1	M3	{M1, M3}				
{}	M3	M1	{M1, M3}				
{}	M2	M3	{M2, M3}				
{}	M3	M2	{M2, M3}				

What Kodkod cannot infer in advance, however, is that the order of allocation is a *total ordering* of the atoms in the domain: each atom allocated is a singleton that has never previously been allocated. Our optimization exploits this knowledge to choose an arbitrary total ordering in advance, by convention the lexical ordering, and constrains the allocations to follow this total order. On the marble example, this optimization reduces the possible executions to just two, exactly one from each equivalence class:

Marble_{old}	a	b	Marble	Marble_{old}	a	b	Marble
$Marble_0$	a_1	b_2	$Marble_0 \cup a_1 \cup b_2$	$Marble_0$	a_1	b_2	$Marble_0 \cup a_1 \cup b_2$
{}	M1	M2	{M1, M2}	M1	M2	M3	{M1, M2, M3}

The optimization requires a change to how the symbolic execution constructs the initial symbolic state and to the symbolic execution rules for create statements. Theses changes are discussed below.

3.4.2 Modified Initial State

Section 3.2.2 listed the relations added to the initial relational declaration, $\mathcal{D}_0$. In addition to those, for every user-defined domain M in the FIR program, the optimized symbolic execution adds two more:

- M_{order}, a binary relation that totally orders the atoms in the scope of M; and
- M_{first}, a singleton set containing the first element in the total order.

For each of these domains M, the optimized symbolic execution also adds a formula to the initial path constraint, $\mathcal{P}_0$, that constrains the initial value of the domain to be some prefix of the total order:

$$M_{order}.M_0 \subseteq M_0$$

which says the join of the total order relation with the set of all initial elements is a subset of those initial elements. This is true if and only if M_0 is a prefix of M_{order}.

In the bounds of the relational logic problem, Forge fixes each relation M_{order} and M_{first} to the lexical ordering. Recall from Section 3.1, that a relation in the logic is declared as follows:

$$relation :_{arity} [lowerBound, upperBound]$$

where the *lowerBound* are the tuples that must be in the relation and the *upperBound* are those that may be in the relation.

If the user chooses a scope on a domain M of, say, 3, then the declaration of M_{order} and M_{first} would bound them both above and below by (exactly to) the following lexical order:

$$M_{order} :_2 [\ \{\langle \mathtt{M}_1, \mathtt{M}_2\rangle, \langle \mathtt{M}_2, \mathtt{M}_3\rangle\}, \{\langle \mathtt{M}_1, \mathtt{M}_2\rangle, \langle \mathtt{M}_2, \mathtt{M}_3\rangle\}]$$
$$M_{first} :_1 [\{\langle \mathtt{M}_1\rangle\}, \{\langle \mathtt{M}_1\rangle\}]$$

This modified initial symbolic state breaks symmetries on the possible initial values of the domains, which is equivalent to what the lex-leader predicate accomplished in our example. The modified inline and constrain rules for create statements, described next, go further to break symmetries on the entire allocation order.

3.4.3 Modified Rules for Create Statements.

A FIR create statement has the following form:

$$\mathsf{v} := \textbf{new}\ \mathsf{M}$$

From a symbolic state $\langle \mathcal{D}, \mathcal{P}, \mathcal{E}\rangle$, the modified symbolic execution rules bind v to an expression that evaluates to the next atom to be allocated from the total order. Let *currs* be $\mathcal{E}(\mathsf{M})$, an expression which evaluates to the set of atoms of type M currently allocated. Since the initial value of M was constrained to be a prefix of the total order, and since these rules allocate the next atom in the total order, we know by induction that *currs* will always be a prefix of the total order as well.

Given that *currs* is a prefix of the total order, let *next* denote the following expression, which evaluates to the next atom to be allocated in the total order:

$$\textbf{no}\ currs\ ?\ M_{first} : (currs.M_{order} \setminus currs)$$

If *currs* is empty — if the domain has no atoms allocated yet — then *next* is the first atom in the total order; otherwise, *next* is obtained by joining the current atoms with the total order and then removing the current atoms. Joining *currs* with the total order shifts the *currs* prefix over by one, so that it contains the next atom but previous atoms as well. Taking the set difference of that expression and *currs* removes the previous atoms, leaving just the next atom.

But what if all of the atoms have been allocated so that there is no next atom? In that case, $currs.M_{order}$ does not contain any additional atom, and *next* evaluates to the empty set. To address this, the modified symbolic execution rules add a formula

to the path constraint forcing *next* to be non-empty. This makes the create statement infeasible when there are no more atoms to be allocated, which was also the case with the original symbolic execution rules and cannot be avoided when working with finite domains.

The modified inline rule for create statements binds the assigned variable to the *next* expression, adds the formula that *next* be non-empty to the path constraint, and binds the domain M to the union of *currs* and *next*:

$$\frac{\mathit{currs} = \mathcal{E}(\mathsf{M}) \qquad \mathit{next} = \textbf{no}\ \mathit{currs}\ ?\ M_{\mathit{first}} : (\mathit{currs}.M_{\mathit{order}} \setminus \mathit{currs})}{\mathsf{v} := \textbf{new}\ \mathsf{M}, \langle \mathcal{D}, \mathcal{P}, \mathcal{E} \rangle \Longrightarrow_I \langle \mathcal{D}, \mathcal{P} \cup \{\textbf{some}\ \mathit{next}\}, \mathcal{E}[\mathsf{v} \mapsto \mathit{next},\ \mathsf{M} \mapsto (\mathit{currs} \cup \mathit{next})] \rangle}$$

The constrain strategy is similar except it declares fresh relations for v and M:

$$\frac{\mathit{currs} = \mathcal{E}(\mathsf{M}) \qquad \mathit{next} = \textbf{no}\ \mathit{currs}\ ?\ M_{\mathit{first}} : (\mathit{currs}.M_{\mathit{order}} \setminus \mathit{currs})}{\mathsf{v} := \textbf{new}\ \mathsf{M}, \langle \mathcal{D}, \mathcal{P}, \mathcal{E} \rangle \Longrightarrow_C \langle \mathcal{D} \cup \{v', M'\}, \mathcal{P} \cup \{\textbf{some}\ v', v' = \mathit{next}, M' = \mathit{currs} \cup v'\}, \mathcal{E}[\mathsf{v} \mapsto v',\ \mathsf{M} \mapsto M'] \rangle}$$

3.4.4 Implications for Correctness

In Section 3.3, we argued for the correctness of the symbolic execution by demonstrating that the symbolic states it constructs encode exactly the set of feasible program bindings. With the optimization, that is no longer true. The optimization has no effect on the soundness of the symbolic states: every logic binding they permit still encodes some feasible program binding. But it does affect the completeness of the symbolic states, because many feasible program bindings in each equivalence class have been excluded from the encoding.

To prove the correctness of the optimized symbolic execution, we have to weaken our definition of completeness to allow for symmetry breaking. The former definition said that a symbolic state $\langle \mathcal{D}, \mathcal{P}, \mathcal{E}, \rangle$ is a complete encoding of a set of program bindings $\mathcal{B}$ if, for every program binding $pb \in \mathcal{B}$, there exists a logic binding lb that is a solution to $\mathcal{P}$ and that encodes pb:

$$\mathit{complete}(\mathcal{D}, \mathcal{P}, \mathcal{E}, \mathcal{B}) \equiv \forall\, pb \in \mathcal{B} \mid \exists\, lb \in \mathit{solutions}^{\mathcal{D}}(\mathcal{P}) \mid \mathit{encodes}(lb, pb, \mathcal{E})$$

Let $\mathit{iso}(pb1, pb2)$ be a predicate on program bindings that is true if the bindings are isomorphic. The modified definition of completeness that accounts for symmetry breaking says that every program binding $pb \in \mathcal{B}$ must be isomorphic to some program binding $pb2$ (possibly itself), such that lb encodes $pb2$:

$$\forall\, pb \in \mathcal{B} \mid \exists\, pb2 \in \mathcal{B}, lb \in \mathit{solutions}^{\mathcal{D}}(\mathcal{P}) \mid \mathit{iso}(pb, pb2) \wedge \mathit{encodes}(lb, pb2, \mathcal{E})$$

The new definition adds some complexity to the proofs of completeness but should not fundamentally alter their structure.

3.5 Chapter Summary

This chapter explained the Forge bounded verification analysis. It showed how the analysis, from a procedure and specification in FIR, uses symbolic execution to derive a formula in relational logic that is true if there exists and execution of the procedure that violates the specification. It also offered an argument for the correctness of the symbolic execution and described an optimization of it that uses symmetry breaking. The next chapter explains the coverage metric that complements the bounded verification analysis.

Usually, when a tool claims to use "symbolic execution," the connotation is that it symbolically executes and analyzes each path through the code individually. It is rare for a tool to perform an "all paths" symbolic execution as we do. In the future, it would be interesting to implement a version of Forge that uses single-path symbolic execution and empirically compare it to our current version. Which is faster to solve: one large SAT problem or several smaller SAT problems? The answer is not clear to us and probably depends on the context. However, if the analysis is performed on a multi-core machine, a single-path symbolic execution would allow those analyses to be parallelized, which could offer a significant performance boost.

Chapter 4

Coverage Metric

The bounded verification analysis described in the previous chapter is unsound. Not being a full verification of the program, it may fail to find counterexamples when they exist. When the bounded verification analysis does not find an existing problem in a procedure, it is because the analysis, in some way, did not explore, or *cover*, all of the procedure's behaviors. The coverage metric described in this chapter attempts to identify statements that were "missed" (not covered) by the bounded verification, so that Forge can, in turn, warn the user of these missed statements. This information can help the user determine whether the bound, the specification, or the code should be modified to make the analysis more comprehensive.

4.1 Examples of Poor Coverage

The coverage metric identifies statements that were "missed" by the bounded verification analysis when the analysis does not find a counterexample. A statement is marked as "missed" if it could be replaced with any statement that modifies at most the same variables — it has the same frame condition — regardless of how it modifies those variables. Missed statements are effectively statements that were not *needed* by the analysis to find the procedure correct (within the bound).

There are four scenarios in which bounded verification may fail to find a problem when one exists. These scenarios are explained below and illustrated by the examples in Figure 4-1. The statements identified as missed by the coverage metric are highlighted in gray in the figure. By identifying these missed statements, the metric can help a user determine when one of these scenarios has occurred.

Under-constrained specification. If the specification of the code is accidentally under-constrained, then the code may satisfy the specification while nevertheless allowing buggy behaviors. At the extreme, the specification may be a tautology, in which case every implementation will vacuously satisfy it.

Example (a) in Figure 4-1 shows how an under-constrained specification can mask bugs. The specification of the swap procedure requires only that x' equals y (and not that y' equals x), and, as a result, the analysis cannot detect the bug. Since the bounded verification would succeed given any assignment to y' at Statement 2, the metric identifies that statement as missed.

code	specification
(a) *under-constrained specification*	
0 **proc** swap(x, y) : (x', y')	x', y' := **spec**(x' = y)
1 x' := y	
2 y' := x'	
3 **exit**	
(b) *over-constrained spec statement*	
0 **proc** registerUser(u: User) : ()	registered, id := **spec**(∀ u1, u2 |
1 registered := registered ∪ u	(u1≠u2 ∧ u1 ∪ u2 ⊆ registered) ⇒
2 id := **spec**(∃ uid | ¬(uid ⊆ User.id)	u1.id ≠ u2.id)
∧ (id = id_{old} ∪ u→uid)	
3 **exit**	
(c) *infinite loop*	
0 **proc** cube(n) : (result)	result := **spec**(result = n × n × n)
1 exponent := 3	
2 result := 1	
3 **while** (exponent > 0)	
4 result := result × n	
5 **exit**	
(d) *insufficient bound*	
0 **proc** putCache(key, value) : ()	cache := **spec**(#cache ≤ 10 ∧
1 **if** (#cache < 10)	key→value ⊆ cache)
2 cache := cache ⊕ key→value	
else	(not covered when the scope
3 cache := **spec**(cache ⊆ $cache_{old}$ ∧	has less than 10 keys)
one $cache_{old}$ - cache)	
4 **exit**	

Figure 4-1: Examples of Poor Coverage. Bounded verification does not find counterexample for these examples, yet problems remain. The statements shown in gray are "missed" (not covered) by the bounded verification analysis.

Over-constrained spec statement. Bounded verification is incapable of detecting accidentally over-constrained code. Because it eliminates possible behaviors, an over-constraint may only make code *more* like to satisfy its specification, not less. At the extreme, the code may have no feasible executions (the final path constraint is a contradiction), which would cause it to vacuously satisfy any specification.

One source of over-constraint is an infeasible specification statement in the procedure. The register procedure in example (b) contains a specification statement, Statement 2, that was intended to find a new id not used in the pre-state but was accidentally written to find an id not used in the post-state; that is, $\neg$(uid $\subseteq$ User.id) should be $\neg$(uid $\subseteq$ User.id$_{old}$). When conjoined with the constraint that forces the id be used in the post state, (id = id$_{old}$ $\cup$ u$\rightarrow$uid), this constraint makes the specification statement infeasible, and the procedure vacuously satisfies its specification.

In this example, the analysis needs only Statement 2 to find the register procedure correct, so the metric finds all other statements to be missed. In fact, because this procedure would have satisfied *any* specification, the metric identifies the specification of the procedure as missed, too.

Infinite loop. Another source of over-constraint is an infinite loop or recursion in the procedure. Bounded verification cannot detect that a procedure might not terminate, because non-terminating executions are not feasible within *any* finite number of loop unrollings. (The unrolling of an infinite loop or infinite recursion introduces an infeasible specification statement in the code, so all over-constraints are ultimately brought about by an over-constrained specification statement.)

In example (c), the developer failed to decrement exponent inside the loop, causing the loop never to terminate. In this example, Statements 1 and 3 are sufficient for non-termination, so all other statements are identified as missed.

Insufficient bound. If the bound on the analysis is too low, buggy behaviors of the code may go undetected. For example, if the code contains a branch that is used only for large data structures, then a low bound could make the statements on that branch irrelevant to the procedure's correctness.

Example (d) in Figure 4-1 contains a bug that would go undetected in too low a bound. When the cache size is less than 10, the given key-value pair is correctly added to the cache (overriding any previous mapping for the key). However, when it is greater than 10, it drops some arbitrary pair from the cache but, incorrectly, does not add the given pair to the cache. If the scope on the keys is set to less than 10, then this bug goes undetected by our analysis. Since cache could be assigned any value at Statement 3 without affecting the analysis, Statement 3 is missed.

The examples in Figure 4-1 demonstrate the usefulness of a coverage metric to a user of the bounded verification analysis. It is important to note, however, that a lack of full coverage is not by itself a definitive sign of a problem with the analysis. For example, a specification may be *intentionally* under-constrained in a way that leaves statements missed but still does not mask some latent bug. Also, statements will be identified as missed if they are dead or redundant code, even though these statements do not cause undesired behavior (but a user may wish to remove them anyway).

4.2 Exploiting the Unsatisfiable Core

To explain how the coverage metric detects missed statements, we first review how the bounded verification analysis works. From the code of a procedure, the analysis obtains by symbolic execution a set of formulas $\mathcal{P}_f$, and from a specification of the procedure, it obtains a formula ψ. It then invokes the Kodkod model finder to find a solution to the formulas in $\mathcal{P}_f \cup \{\neg\psi\}$. If Kodkod finds such a solution, the analysis uses it to construct a trace of the procedure that violates the specification.

When a model finder determines a set of formulas to be satisfiable, it always reports evidence of the satisfiability to the user: namely, the solution itself. However, when the formulas are unsatisfiable, most model finders are incapable of reporting any feedback as to the cause of the unsatisfiability [82]. The Kodkod model finder stands out in this respect. When given an unsatisfiable set of formulas, Kodkod can extract an *unsatisfiable core* [83], a subset of the formulas that is by itself unsatisfiable. The core reported by Kodkod is locally minimal: removing any single formula from the core renders it satisfiable.

Let the final path constraint $\mathcal{P}_f$ derived by the symbolic execution be $\{f_1, \ldots, f_k\}$. The set of formulas handed to Kodkod is, therefore, $\{f_1, \ldots, f_k, \neg\psi\}$, of which the unsatisfiable core will be some subset. If $\neg\psi$ is in the core, then the metric identifies the specification as covered. But if a formula f_i is in the core, which statements in the code should be considered covered? Namely, which statements are responsible for f_i being in the path constraint?

To answer this question, we augment the symbolic execution so that, in addition to $\mathcal{D}$, $\mathcal{P}$, and $\mathcal{E}$, it maintains a *formula slice map*, $\mathcal{F}$, which maps each formula, upon its insertion into the path constraint, to the set of statements in (or "slice" of) the procedure from which that formula was generated:

$$\mathcal{F} : \mathcal{P} \rightarrow 2^{Stmt}$$

In addition to $\langle \mathcal{D}_0, \mathcal{P}_0, \mathcal{E}_0 \rangle$, the augmented symbolic execution begins from an initial formula slice map, F_0, that is empty; and the result of the symbolic execution includes, in addition to $\langle \mathcal{D}_f, \mathcal{P}_f, \mathcal{E}_f \rangle$, a final formula slice map $\mathcal{F}_f$. If the unsatisfiable core is $\mathcal{C}$, then the metric considers the following statements to be covered:

$$\bigcup_{c \in \mathcal{C}} \mathcal{F}_f(c)$$

and the rest are identified as missed.

For the coverage metric to be sound in its detection of missed statements — for it to only mark statements as missed when they are not needed for correctness — the symbolic execution must use the constrain strategy. Since the inline strategy avoids adding formulas to the path constraint, it prevents the metric from finding statements to be covered when in fact they were. The next section explains how the formula slice map is updated by the symbolic execution, shows the technique on an example program, and illustrates the problem with using the inline strategy.

4.3 Symbolic Execution with Formula Slicing

When symbolically executing update statements — assign, create, or specification statements — updating the formula slice map is straightforward. Recall that applying the constrain rule to an assignment statement S of the form $\mathsf{v} := \mathsf{e}$, starting from the symbolic state $\langle \mathcal{D}, \mathcal{P}, \mathcal{E} \rangle$, yields a state $\langle \mathcal{D}', \mathcal{P}', \mathcal{E}' \rangle$, where $\mathcal{P}'$ was the following form:

$$\mathcal{P}' = \mathcal{P} \cup \{v' = \mathcal{T}_R[\![\mathsf{e}, \mathcal{E}]\!]\}$$

When the coverage metric is enabled, the symbolic execution of this assignment statement S adds to the current formula slice map, $\mathcal{F}$, a mapping from the formula added to the path constraint to the singleton set containing S:

$$\mathcal{F}' = \mathcal{F} \cup \{(v' = \mathcal{T}_R[\![\mathsf{e}, \mathcal{E}]\!]) \mapsto \{\mathsf{S}\}\}$$

Now if the formula $v' = T_R[\![\mathsf{e}, \mathcal{E}]\!]$ ends up in the unsatisfiable core, it will be mapped back to S, and S will be considered covered. Create statements and specification statements are handled in the same way: each formula they add to the path constraint is bound in the formula slice map to the singleton set containing the statement.

Updating the formula slice map when symbolically executing branch statements is more involved. Recall that upon encountering a branch statement of the form **if** e **then** S_T **else** S_F, the symbolic execution generates from S_T and S_F the respective symbolic states $\langle \mathcal{D}_T, \mathcal{P}_T, \mathcal{E}_T \rangle$ and $\langle \mathcal{D}_F, \mathcal{P}_F, \mathcal{E}_F \rangle$, and then merges these states into a single symbolic state $\langle \mathcal{D}', \mathcal{P}', \mathcal{E}' \rangle$, where $\mathcal{P}'$ has the following form:

$$\begin{aligned}\mathcal{P}' = \mathcal{P} \cup \{cond \Rightarrow f \mid f \in (\mathcal{P}_T \setminus \mathcal{P})\} \cup \{\neg cond \Rightarrow f \mid f \in (\mathcal{P}_F \setminus \mathcal{P})\} \cup \\ \{cond \Rightarrow v' = \mathcal{E}_T(\mathsf{v}) \mid \mathsf{v} \in \mathcal{V}_B\} \cup \{\neg cond \Rightarrow v' = \mathcal{E}_F(\mathsf{v}) \mid \mathsf{v} \in \mathcal{V}_B\}\end{aligned}$$

where *cond* is again the translation of the branch condition to a logic formula, and $\mathcal{V}_B$ are the variables modifid by one or both sides of the branch. (Recalling the precise definition of $\mathcal{V}_B$ isn't needed to understanding how the formula slice map is updated.)

Our augmented symbolic execution will produce, in addition to $\langle \mathcal{D}_T, \mathcal{P}_T, \mathcal{E}_T \rangle$ and $\langle \mathcal{D}_F, \mathcal{P}_F, \mathcal{E}_F \rangle$, two formula slices maps $\mathcal{F}_T$ and $\mathcal{F}_F$. Merging $\mathcal{F}_T$ and $\mathcal{F}_F$ for a branch statement S yields the following formula slice map after the statement:

$$\begin{aligned}\mathcal{F}' = \mathcal{F} \cup \{(cond \Rightarrow f) \mapsto (\mathcal{F}_T(f) \cup \{\mathsf{S}\}) \mid f \in (\mathcal{P}_T \setminus \mathcal{P})\} \cup \\ \{(\neg cond \Rightarrow f) \mapsto (\mathcal{F}_F(f) \cup \{\mathsf{S}\}) \mid f \in (\mathcal{P}_F \setminus \mathcal{P})\} \cup \\ \{(cond \Rightarrow v' = \mathcal{E}_T(\mathsf{v})) \mapsto \{\mathsf{S}\} \mid \mathsf{v} \in \mathcal{V}_B\} \cup \\ \{(\neg cond \Rightarrow v' = \mathcal{E}_F(\mathsf{v})) \mapsto \{\mathsf{S}\} \mid \mathsf{v} \in \mathcal{V}_B\}\end{aligned}$$

Each of the formulas of the form $(cond \Rightarrow f)$ is the result of both the statements that generated f (all the statements in $\mathcal{F}_T(f)$) and the statement that generated *cond* (the branch S itself). Similarly, each formula $\neg cond \Rightarrow f$ is the result of both $\mathcal{F}_F(f)$ and S. The remaining formulas added to the path constraint — all those of the form $(cond \Rightarrow v' = \mathcal{E}_T(\mathsf{v}))$ and $(\neg cond \Rightarrow v' = \mathcal{E}_F(\mathsf{v}))$ — are generated soley by S itself.

An example of this formula slicing process is shown next.

```
proc register (newEmail) : (newUser)
1   if newEmail ⊆ User.email
2     newUser := Error
    else
3     newUser := new User
4     email := email ∪ newUser→newEmail
5     id := spec(∃ newId | ¬(newId ⊆ User.id_old) ∧
                           id = (id_old ∪ newUser→newId));
    exit
```

Figure 4-2: Register procedure

4.3.1 Example: Coverage of the Register Procedure

To illustrate the coverage metric, consider the register procedure in Figure 4-2. Unlike the version of the procedure in Chapter 3, this one satisfies its specification, so the bounded verification analysis finds no counterexamples, regardless of the bound.

The formula slice map that results from executing the true side of the branch, $\mathcal{F}_T$, will map the formula added to the path constraint by Statement 2, call it f_2, to the singleton set containing Statement 2, abbreviated here S_2:

$$\mathcal{F}_T = \{f_2 \mapsto \{S_2\}\}$$

Similarly, the formula slice map from the false side of the branch maps each of the formulas generated from statements 3, 4, and 5 to the singleton sets containing those statements[1]:

$$\mathcal{F}_F = \{f_3 \mapsto \{S_3\}, f_4 \mapsto \{S_4\}, f_5 \mapsto \{S_5\}\}$$

Merging the sides of the branch yields the final formula slice map:

$$\mathcal{F}_f = \{(cond \Rightarrow f_2) \rightarrow \{S_1, S_2\}, (\neg cond \Rightarrow f_3) \rightarrow \{S_1, S_3\},$$
$$(\neg cond \Rightarrow f_4) \rightarrow \{S_1, S_4\}, (\neg cond \Rightarrow f_5) \rightarrow \{S_1, S_5\}, f_1 \rightarrow \{S_1\}\}$$

Since S_2 generated f_2 and S_1 generated $cond$, they are both responsible for $cond \Rightarrow f_2$. Similarly, the path constraints from the false side of the branch of the form $\neg cond \Rightarrow f_i$ are generated by both the statement S_i and S. Lastly, f_1 is the formula added to the path constraint by the branch S_1 itself.

The specification of register for this example is the same as before:

$$\text{newUser, email, id} := \textbf{spec}(\neg(\text{newEmail} \subseteq \text{User.email}_{old}) \Rightarrow$$
$$(\text{newUser.email} = \text{newEmail} \wedge \neg(\text{newUser.id} \subseteq \text{User.id}_{old})))$$

[1]These statements can add multiple formulas to the path constraint, but this description generalizes naturally to that case. Plus, we can equivalently presume that rather than adding multiple formulas, they add a single conjunction of those formulas.

Because the procedure meets this specification, bounded verification does not find any counterexamples, but Kodkod reports the following unsatisfiable core:

$$\{f_1, \neg cond \Rightarrow f_3, \neg cond \Rightarrow f_4, \neg cond \Rightarrow f_5, \neg\psi\}$$

From the presence of $\neg\psi$ in the core, the metric finds the specification of the register procedure to be covered. Looking up the remaining formulas in $\mathcal{F}_f$ identifies the following statements as covered:

$$\{\mathsf{S}_1, \mathsf{S}_3, \mathsf{S}_4, \mathsf{S}_5\}$$

but S_2 as missed. Indeed, the specification does not stipulate what must happen when the branch condition, $\mathsf{newEmail} \subseteq \mathsf{User.email}_{old}$, is true, so the assignment in Statement 2, $\mathsf{newUser} := \mathsf{Error}$, does not affect the correctness of the procedure.

This is an example of where a lack of full coverage may not be indicative of a problem with the analysis. In this case, the user may have intentionally left the specified behavior non-deterministic when given a duplicate email. In either case, the coverage metric provides additional information that will help the user make that determination.

4.3.2 The Problem with the Inline Strategy

This formula slicing technique does not produce an accurate coverage measure when using the inline strategy for symbolic execution. Because the inline strategy avoids adding formulas to the path constraint, mapping formulas in the path constraint back to the statements from which they were generated does not adequately track the impact of statements. Namely, it can mark statements as missed when they were in fact necessary for correctness.

For example, when the inline strategy is applied to the register procedure in Figure 4-2, Statements 2 and 4 do not introduce any additional formulas to the path constraint. Following the coverage technique as described above, the final formula slice map $\mathcal{F}_f$ would be:

$$\mathcal{F}_f = \{(\neg cond \Rightarrow f_3) \rightarrow \{S_1, S_3\}, (\neg cond \Rightarrow f_5) \rightarrow \{S_1, S_5\}\}$$

And the core reported by Kodkod would include the following formulas:

$$\{\neg cond \Rightarrow f_3, \neg cond \Rightarrow f_5, \neg\psi\}$$

Thus, the analysis would correctly find the specification and statements $\{\mathsf{S}_1, \mathsf{S}_3, \mathsf{S}_5\}$ to be covered, but it would incorrectly identify S_4 as missed, even though a counterexample would be found if it were removed.

To prevent covered statements from being identified as missed when using the inline strategy, a number of over-approximation techniques could be incorporated into the metric that finds statements to be covered even when they do not correspond to formulas in the core. We briefly experimented with an over-approximation technique that used variable slicing in addition to formula slicing, but it proved too imprecise to be useful in practice, often marking the entire procedure as covered when many statements were not needed for correctness.

4.4 Coverage before Unrolling

Recall that the symbolic execution is performed on the procedure after it has been unrolled. As a result, the coverage technique just described measures the coverage of the procedure after unrolling. A question remains of how this coverage of the unrolled procedure should be mapped to a coverage measure of the original FIR procedure.

To investigate this question, we consider the following FIR program whose **length** procedure correctly calculates the length of a linked list and logs that it has been called:

```
00  domain Node
01  global head: Node, global next: Node→Node, global log: Boolean
02  local curr: Node, local len: Integer
03
04  proc length () : (len)
05    log := true
06    len := 0
07    curr := head
08    while (some curr)
09      len := len + 1
10      curr := curr.next
```

And consider an analysis in a bound of 2 **Nodes**, 2 loop unrollings, and a bitwidth of 3 (integers -4 to 3) that checks whether the procedure meets the following specification:

```
log, len := spec(len = #head.*next)
```

which says that the length of the list is equal to the number of nodes reachable from the head of the list and that the log may change arbitrarily.

The bounded verification analysis first unrolls the loop in the procedure twice and then performs a symbolic execution to search for counterexample traces to the unrolled procedure. Since the procedure (original and unrolled) meets the specification, no counterexamples are found. When the coverage metric is enabled, it finds the gray statements in the unrolled procedure below to be missed:

```
11  proc length () : (len)
12    log := true
13    len := 0
14    curr := head
15    if (some curr)
16      len := len + 1
17      curr := curr.next
18      if (some curr)
19        len := len + 1
20        curr := curr.next
21        spec(no curr)
```

Statement 12 is missed due to an under-constrained specification that puts no requirements on the final value of the log variable. Statements 20 and 21 are missed due to an insufficient bound: in lists of length two nodes or less (the bound chosen for the analysis), the expression curr.next at Statement 20 will always be the empty set, so that assignment and the subsequent constraint imposed by the specification statement are unnecessary.

From this information, Forge can present the coverage of the original procedure in two formats: on the left, a fine-grained view of the coverage, and on the right, a summary view:

```
04   proc length () : (len)
05   (0/1)  log := true
06   (1/1)  len := 0;
07   (1/1)  curr := head;
08   (2/3)  while (some curr)
09   (2/2)     len := len + 1;
10   (1/2)     curr := curr.next;
```

```
04   proc length () : (len)
05      log := true
06      len := 0
07      curr := head
08      while (some curr)
09         len := len + 1
10         curr := curr.next
```

In the fine-grained view, each statement is given a fractional score. The denominator is the number of statements in the unrolled procedure to which that statement corresponds; and the numerator is the number of those corresponding statements that were covered. The while-statement (Statement 8), for example, corresponds to three statements in the unrolled procedure: two branches (Statements 15 and 18) that were covered and a specification statement (Statement 21) that was missed. In the summary view, a statement is highlighted as missed if its numerator is zero, i.e., if all of its corresponding statements in the unrolled procedure were missed, and the fractions are not shown.

The advantage of the fine-grained view is that it can show a user when the bound was not sufficient to fully explore all the loop unrollings. Here, it shows that a scope of 2 nodes is insufficient to fully explore three loop unrollings — a scope of 3 is actually needed to falsify the loop condition at the end of the second trip through the loop. The summary view shows the statement missed due to an under-constrained specification, but its purely binary notion of coverage — each statement is either missed or not — cannot reveal the insufficient bound.

However, in our experience, users tend to find the fine-grained view confusing, mainly because it requires an understanding of how the unrolling works. With this example, for instance, users were often confused as to why the denominator of the while-loop is three when only two unrollings were specified. The fine-grained view also has the non-intuitive property that increasing the number of loop unrollings can *decrease* the fraction to which a statement has been covered, because more unrollings means more opportunities for statements in the unrolled procedure to be missed.

Currently, the Forge tool has been configured to provide only the summary view of coverage by default, with the fine-grained view still available for experts. Since the summary view is therefore the one that will be used most often, it is the one we use when evaluating the effectiveness of the coverage metric in Chapter 6.

4.5 Chapter Summary

This chapter presented the coverage metric used by Forge. It discussed the scenarios in which poor coverage by the analysis can mask a problem in the code, and it explained how the coverage is calculated by mapping the formulas in the unsatisfiable core back to the statements that generated them. The next chapter addresses two challenges that arise when applying Forge to the analysis of programs written in high-level languages.

While the Forge coverage metric tells the user the statements that were missed by the analysis, it does not attempt to explain *why* those statements were missed. It would be interesting to explore in future work whether the tool could provide a helpful answer to this "why" question. For example, if the specification of the procedure is missed by the analysis, the tool might at least report that "the procedure has no executions within the provided bound." Or, if the body of a loop is missed but the loop condition is not, the tool could warn that the "loop may not terminate." Like this infinite loop warning, some reports could result from heuristic pattern-matching for common problems. Although not definitive diagnoses, such warnings could nevertheless be useful.

Chapter 5

Object Orientation and Data Abstraction

Two key challenges arise when attempting to apply the Forge framework to the analysis of programs written in high-level, object oriented (OO) programming languages. The first challenge is the task of translating the code of a high-level program to the Forge Intermediate Representation. Section 5.1 of this chapter explains the key ideas involved in encoding OO programs in FIR.

The second challenge arises from the specifications of abstract datatypes. Recall that the bounded verification analysis automatically obtains a logic formula $P(s, s')$ that constrains the relationship between a pre-state s and a post-state s' and that holds whenever an execution exists from s that terminates in s'. A second formula $\psi(s, s')$ is obtained from a user-provided specification, and the analysis searches for counterexamples to the following *correctness claim*:

$$\forall s, s' \mid P(s, s') \Rightarrow \psi(s, s')$$

In this formula, the procedure and specification are both in terms of the same pair of states, s and s'. When checking a procedure of an abstract data type, however, the (abstract) representation referred to in the specification usually differs from the (concrete) representation in the code. The standard technique for bridging the gap involves the user's providing an abstraction predicate $\alpha(c, a)$ for the data type that relates concrete and abstract states, and a specification $S(a, a')$. Section 5.2 below explains how to formulate the claim that the procedure of an abstract datatype satisfies its specification in a form that is tractable for our bounded verification.

5.1 From Object-Oriented Code to FIR

The translation of OO programs to FIR is based on a *relational view of the heap* [47], in which program constructs are interpreted as relations or operations on relations. Specifically, types are viewed as sets; fields as binary, functional relations that map elements of their class to elements of their target type; and local variables as singleton

sets. In this setting, field dereference becomes relational join and field update becomes relational override.

Under the relational view of the heap, field dereference can be encoded as relational join. To illustrate, consider the field dereference `x.value`. The relational view of the heap encodes the local variable `x` as a singleton set, call it rx, and the field `value` as a functional relation, call it rvalue. Recall that since rx is singleton and rvalue a function, the join expression rx.rvalue is equivalent to function application and yields a singleton set. This singleton set represents the value of the dereference `x.value`.

An update to a non-static field is encoded using a relational override ($\oplus$) operator. The field update `this.value = v` is encoded in FIR as the assignment value := value $\oplus$ this→v, where value $\oplus$ this→v evaluates to the set of tuples containing the tuple this→v and any tuples in value that do not begin with this.

5.1.1 An Example Translation

The basic process of translating OO code to FIR will be illustrated here by example, starting from the Java code in Figure 5-1 and resulting in the FIR program in Figure 5-2. The example omits some of the complexities that arise in real high-level programs, including exceptions and dynamic dispatch, but solutions to these are explained below in Section 5.1.2 and are implemented in the JForge front-end.

The translation associates each class and interface in the type hierarchy of the OO program with a type in FIR. To do so, it uses a technique called atomization [33], which encodes each type as a union of disjoint FIR domains. For each concrete (non-abstract) class in the hierarchy, atomization creates a corresponding FIR domain, so for the birthday program in Figure 5-1, it declares three domains: Birthday, Month, and Object (Figure 5-2, Line 00). A class's corresponding domain represents the set of objects whose *runtime type* is that class. That is, the Object domain doesn't represent the set of all instances whose Java static type is `Object`, a set which would include all `Birthday` and `Month` instances too, but only those instances of type `Object` that do not belong to any subtype.

Each *static type* in the OO program is then associated with the FIR type that is the union of all FIR domains that correspond to concrete subtypes of that static type. For our example, the translation associates the static types in Figure 5-1 to FIR types with the following mapping:

`Birthday`	$\mapsto$	Birthday
`Month`	$\mapsto$	Month
`Object`	$\mapsto$	Birthday $\cup$ Month $\cup$ Object

The static types `Birthday` and `Month` are mapped to the FIR domains Birthday and Month respectively. Because every Java class is a subclass of `Object`, the translation maps the static type `Object` to the FIR union type Birthday $\cup$ Month $\cup$ Object.

```
class Birthday {
    @NonNull Month month;
    int day;

    @Requires("d > 0 && d <= this.month.maxDay")
    @Modifies("this.day")
    @Ensures("this.day = d")
    void setDay(int d) {
        Month m = this.month;
        boolean dayOk = m.checkDay(d);
        if (dayOk)
            this.day = d;
    }
}

class Month {
    int maxDay;

    @Ensures("return <=> (d > 0 & d <= this.maxDay)")
    boolean checkDay(int d) {
        return false;
    }
}
```

Figure 5-1: Birthday Example in Java

```
00  domain Birthday, domain Month, domain Object
01  global month: Birthday → Month
02  global day: Birthday → Integer
03  global maxDay: Month → Integer
04  local this: Birthday, local d: Integer
05  local m: Month, local dayOk: Boolean
06
07  proc setDay (this, d) : ()
08    m := this.month
09    dayOk := spec (dayOk ⇔ (d > 0 ∧ d ≤ m.maxDay))
10    if dayOk
11      day := day ⊕ (this → d)
12    exit
```

Figure 5-2: Translation of `Birthday.setDay` into FIR

The translation encodes each OO field as a global variable that maps members of the enclosing class to members of the field's type. For example, the field `month` in Figure 5-1 is encoded as a FIR global month whose type is Birthday → Month (Line 01). Similarly, the translation creates global variables day: Birthday → Integer and maxDay: Month → Integer (Lines 02-03). The translation will add constraints that these binary relations be functions to the analysis of any method. For each parameter and local variable in the method under analysis, the translation declares a local variable of the corresponding type, whose value is constrained to be a scalar. For the `setDay` method, the translation creates four FIR locals: this of type Birthday, d of type Integer, m of type Month, and dayOk of type Boolean (Lines 04-05).

The result of translating the `setDay` method is the FIR setDay procedure, which has two inputs – this and d – and no outputs (Line 07). The procedure begins by assigning the FIR expression this.month to the local variable m (Line 08). Although the FIR statement looks nearly identical to its OO counterpart, the dot (.) operator in FIR stands for relational join, not field dereference. As mentioned above, representing fields as functional relations and locals as singleton sets allows field dereference to be encoded as relational join.

The call to `checkDay` in the OO code has been encoded in FIR as a *specification statement* (Line 09) that is an instantiation of the specification of `checkDay`. As discussed in Section 2.1.2, using specification statements in place of calls makes the analysis of a high-level program *modular*. That is, `setDay` will be analyzed assuming that `checkDay` conforms to its specification, regardless of its implementation. This modularity, in addition to allowing each procedure to be checked in isolation, is crucial to making the analysis scale to real programs.

A field update is encoded using the FIR relational override (⊕) operator. The value of the FIR expression day ⊕ (this → d) is the relation containing the tuple (this → d) and any tuples in day that do not begin with this. Thus, the assignment day = day ⊕ (this → d) (Line 11) encodes the Java statement `this.day = d`. Because the day relation encodes the mapping from *every* `Birthday` to its `day` field, and because this field update is encoded as an assignment to the *entire* relation, this translation to FIR fully captures the effect of any object aliasing.

5.1.2 Complexities of Real Programs

There are a number of additional complexities that arise in object-oriented code that were omitted in the above example. Our solutions to these complexities, as they have been implemented in JForge, are presented below.

Dynamic Dispatch. As shown above, a method invocation is converted into a specification statement that instantiates the specification of that method. However, if the method call is virtual and may dynamically dispatch to multiple targets, which target specification should be used? JForge uses the method specification from the *static* type of the receiver. Note that if the analysis were not modular, then this solution would not be adequate, because some methods are *pure virtual*, i.e., they

have no implementation in the static type. Any abstract method in Java, including all methods of interfaces, are pure virtual.

Exceptions. FIR offers no special constructs for raising exceptions or handling them. To encode the possibility of an exceptional return value, each generated procedure is given an extra output variable named "throw" that stores the value of any exceptional return, and all exception handling is encoded as branches in the control-flow graph. Every statement in code that throws an exception is converted into an assignment to the throw variable, followed by a series of branches that test which of the enclosing `try` blocks will catch the exception. If throw does not match any of the caught types, the FIR code branches to the exit statement of the procedure. To model the propagation of an exception up the call chain, a similar series of branches is introduced after each procedure call.

JForge uses an unsound optimization in its encoding of exceptions that treats each exception class as a singleton. To do so, it creates exactly one FIR literal that by itself represents every `NullPointerException` instance, one literal for `IllegalArgumentException`, and so on; and each dynamic allocation of an exception class is replaced with its literal. As a result, the scope on each exception class can be set to 1, the state space of the analysis is reduced, and performance is improved. However, if the correctness of a Java method depends not only upon the type of exceptions, but upon their fields, then this optimization can theoretically produce spurious counterexamples or additional missed counterexamples; but this is rare in practice.

Collections. JForge provides abstract datatype specifications for common library collections, including sets, maps, and lists. Using abstractions of these collections in place of their concrete representations reduces the complexity of the resulting FIR program to be analyzed, thereby improving the performance of the analysis.

`Sets` are modelled abstractly by a binary relation whose domain is the Java `Set` objects and whose range is the elements in the set. A map is abstracted with a ternary relation that contains ⟨Map, key, value⟩ tuples. The ternary relation is constrained to ensure there is exactly one value for each ⟨map, key⟩ pair. Lists and arrays are abstracted by ternary relations containing ⟨List → index → value⟩ tuples. The indices into a list are constrained to start at zero and continue in consecutive order. That is, a list cannot have a value at index 3 unless it also has a value at index 2. The Java specifications of these collections can be modelled precisely using these relations, so this abstraction does not introduce any unsoundness into the analysis.

5.1.3 Unsoundness and Incompleteness

Although a bounded verification of FIR itself never issues false alarms and always finds a counterexample if one exists within the bounds, front-ends that encode high-level languages in FIR are free to take liberties with their translation that ultimately permit spurious counterexamples to the original program or allow counterexamples within the bound to be missed. JForge takes just such a liberty when it treats exception classes as singletons, as discussed above.

Another potential source of false alarms in JForge is the choice of an integer bitwidth that is less than the true width of Java integers. Consider, for instance, an analysis with a bitwidth of 5 that produces a counterexample due to integer overflow. Because Java integers have a width larger than 5, this counterexample does not represent an actual trace of the code. That said, in our experience, errors due to overflow in a low bitwidth have usually indicate the presence of an analogous error in the true bitwidth. In other words, if the code can overflow at a bitwidth of 5, it can probably overflow at 32. Nevertheless, an analogous counterexample is not guaranteed to exist in the true bitwidth.

5.2 Dealing with Abstraction

As discussed in Chapter 3, from a procedure in the Forge Intermediate Representation, the symbolic execution automatically obtains a formula $P(s, s')$ that holds whenever an execution of the procedure exists from s that terminates in s'. A second formula $\psi(s, s')$ is obtained from a user-provided specification. Using P and ψ we state the *correctness claim*, a formula that is true when every trace of the procedure satisfies the specification.

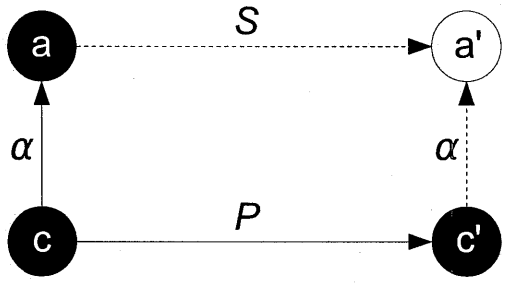
$$\forall s, s' \mid P(s, s') \Rightarrow \psi(s, s')$$

Since the unbounded quantification over those states requires a higher-order, intractable analysis [55], the claim is negated and Skolemized, yielding the *refutation formula*, which is free of higher-order quantifiers and, therefore, tractable:

$$P(s, s') \wedge \neg\psi(s, s')$$

Forge invokes the Kodkod model finder to search for solutions to this formula, which are counterexamples to the correctness claim.

In the formulas above, the procedure and specification are both in terms of the same pair of states, s and s'. When checking a procedure of an abstract data type, however, the (abstract) representation referred to in the specification usually differs from the (concrete) representation in the code. The standard technique for bridging the gap involves the user's providing an abstraction predicate $\alpha(c, a)$ for the data type that relates concrete and abstract states, and a specification $S(a, a')$ for the procedure that relates abstract pre- and post-states [16].



The correctness of the procedure with respect to the specification and abstraction predicate can be established with a forward simulation refinement [88], as visualized by the commuting diagram above. Given the existence of concrete states c and c' related by the procedure P, and given the existence of abstract state a related to c by the abstraction predicate α, the procedure is correct if there exists an abstract state a' such that a and a' are related by the specification S and c' and a' are related by α. We express this constraint in the *refinement-correctness claim*:

$$\forall c, c', a \mid (\alpha(c, a) \wedge P(c, c')) \Rightarrow \exists a' \mid \alpha(c', a') \wedge S(a, a')$$

As we did with the original correctness claim, we can try to eliminate the quantifier in the refinement-correctness claim by negating it and Skolemizing it, which yields the following *refinement-refutation formula*:

$$\alpha(c, a) \wedge P(c, c') \wedge \neg\exists a' \mid \alpha(c', a') \wedge S(a, a') \tag{5.1}$$

Yet, unlike before, after Skolemization, we are still left with a quantifier over an abstract state and, therefore, a higher-order, intractable analysis.

5.2.1 Abstraction Function & Representation Invariant

To make the analysis tractable, we begin by recognizing that in practice the abstraction predicate α is given as the conjunction of two predicates, a representation invariant $R(c)$ on a concrete state, and an abstraction function $A(c, a)$ that interprets concrete states as abstract ones [66]: $\alpha(c, a) \equiv R(c) \wedge A(c, a)$.

Substituting the definition of α into the refinement-refutation formula (5.1) yields:

$$R(c) \wedge A(c, a) \wedge P(c, c') \wedge \neg\exists a' \mid R(c') \wedge A(c', a') \wedge S(a, a')$$

Because $R(c')$ does not depend on a', it can be lifted out of the quantification:

$$R(c) \wedge A(c, a) \wedge P(c, c') \wedge \neg(R(c') \wedge \exists a' \mid A(c', a') \wedge S(a, a')) \tag{5.2}$$

Although the still quantifier persists for now, the following sections will demonstrate that it can be eliminated by assuming that A is a function over the domain of concrete states c that satisfy $R(c)$, a property that nearly always holds in practice. We show that under this assumption, formula 5.2 has a solution if and only if one of the following two formulas has a solution:

- the *invariant preservation formula*:

$$R(c) \wedge P(c, c') \wedge \neg R(c') \tag{5.3}$$

- the *trace inclusion formula*:

$$R(c) \wedge A(c, a) \wedge P(c, c') \wedge A(c', a') \wedge \neg S(a, a') \tag{5.4}$$

Because these two formulas are free of higher-order quantifiers, we will have derived a tractable way to solve the refinement-refutation formula 5.2, providing our assumption that A is a total function over R is correct. If A is not at least a partial function, then solving trace-inclusion could produce spurious solutions to formula 5.2. If A is not total, then solving invariant-preservation and trace-inclusion may miss solutions to formula 5.2.

To avoid spurious counterexamples, we can run a separate check that A is a partial function over R. Finding a solution to the following formula establishes that A is *not* functional:

$$R(c) \wedge A(c, a1) \wedge A(c, a2) \wedge a1 \neq a2$$

While we cannot create a tractable check that A is total over R, we can at least check that it is consistent, i.e., true at least once on the domain R. Finding a solution to the following formula establishes that A is consistent:

$$R(c) \wedge A(c, a)$$

If no solutions are found to the consistency check, then either A is false for all concrete states satisfying R (within the bounds of the analysis) or R itself is false. Both cases indicate an error in the data abstraction.

In order to split the refinement refutation formula into the separate trace-inclusion and invariant preservation formulas, we will need the tools of the next section.

5.2.2 Invariant Preservation and Trace Inclusion

To review, we negated the refinement-correctness claim and substituted $R(c) \wedge A(c, a)$ for $\alpha(c, a)$ to arrive at the refinement-refutation formula:

$$R(c) \wedge A(c, a) \wedge P(c, c') \wedge \neg(R(c') \wedge \exists a' \mid A(c', a') \wedge S(a, a')) \tag{5.5}$$

To eliminate the higher-order quantifier, we need the following lemmas:[1]

Lemma 1.
$\exists x \mid f(x) \wedge g(x)$ and $\forall x \mid f(x) \Rightarrow g(x)$ are equivalent when $\exists! x \mid f(x)$

Proof. Under the assumption that there exists exactly one x such that $f(x)$ is true, we can equivalently name that x, call it e, and rephrase the condition $(\exists! x \mid f(x))$ as the conjunction of two conditions: $f(e) \wedge (\forall x \mid f(x) \Rightarrow x = e)$. Under this condition, both $\exists x \mid f(x) \wedge g(x)$ and $\forall x \mid f(x) \Rightarrow g(x)$ simplify to $g(e)$. □

Lemma 2.
$d \wedge \exists x \mid f(x) \wedge g(x)$ and $d \wedge \forall x \mid f(x) \Rightarrow g(x)$ are equivalent when $d \Rightarrow \exists! x \mid f(x)$

Proof. When d is false, both formulas are false. When d is true, they are equivalent under Lemma 1. □

[1] $\exists! x \mid f(x)$ means there is exactly one x for which $f(x)$ is true.

The assumption that A is a function over R is expressed as follows:

$$\forall c \mid R(c) \Rightarrow \exists! a \mid A(c, a)$$

So for any given c, say c', the following must be true:

$$R(c') \Rightarrow \exists! a' \mid A(c', a')$$

We now apply Lemma 2 to this formula. By letting $d = R(c')$, $f(x) = A(c', x)$, and $g(x) = S(a, x)$, from Lemma 2 we learn that the following two formulas are equivalent:

$$R(c') \wedge \exists a' \mid A(c', a') \wedge S(a, a')$$
$$R(c') \wedge \forall a' \mid A(c', a') \Rightarrow S(a, a')$$

Substituting the latter for the former in the refinement-refutation formula (5.5) yields:

$$R(c) \wedge A(c, a) \wedge P(c, c') \wedge \neg(R(c') \wedge \forall a' \mid A(c', a') \Rightarrow S(a, a'))$$

By pushing negation to the leaves of the formula with DeMorgan's laws, we obtain:

$$R(c) \wedge A(c, a) \wedge P(c, c') \wedge (\neg R(c') \vee \exists a' \mid A(c', a') \wedge \neg S(a, a'))$$

And by distributing over the disjunction we obtain:

$$(R(c) \wedge A(c, a) \wedge P(c, c') \wedge \neg R(c')) \vee \tag{5.6}$$
$$(R(c) \wedge A(c, a) \wedge P(c, c') \wedge \exists a' | A(c', a') \wedge \neg S(a, a')) \tag{5.7}$$

A disjunction is satisfiable if and only if either (or both) of its operands are satisfiable. Thus, we can treat each half of the disjunction as a separate problem.

On formula 5.6, we reuse our assumption that A is a function, and therefore total, over R, to eliminate A from the formula, arriving at the promised *invariant-preservation formula*:

$$R(c) \wedge P(c, c') \wedge \neg R(c')$$

On formula 5.7, we apply Skolemization to yield the promised *trace-inclusion formula*:

$$R(c) \wedge A(c, a) \wedge P(c, c') \wedge A(c', a') \wedge \neg S(a, a')$$

In conclusion, we began with a the refinement-refutation formula, a formula that, if satisfied, represents a counterexample to the claim that the procedure obeys a specification over abstract states, but solving it would have required a higher-order intractable analysis. Using the assumption that A is a function over R, we split the refinement-refutation formula into two separate formulas, (1) invariant preservation and (2) trace inclusion, both free of higher-order quantifiers.

Chapter 6

Case Studies

This chapter reports on three case studies we conducted with the Forge framework and the JForge front-end. In the first study, we checked a series of linked list implementations against an abstract list specification [27]. In the second study, we analyzed electronic voting software [28]. The third study compares the symbolic execution strategies and evaluates the coverage metric on a series of benchmark problems.

6.1 Case Study 1: Linked Lists

This study analyzed a series of linked-list implementations against the Java `List` interface. The implementations were drawn from the Java JDK, the GNU Trove library, the Apache Commons-Collections library, and variants of the JDK implementation that had been seeded with bugs for an MIT software-engineering class.

The Forge framework was used to check twelve methods of each linked list implementation: `add(int, Object)`, `add(Object)`, `clear`, `contains`, `get`, `indexOf`, `isEmpty`, `lastIndexOf`, `remove(int)`, `remove(Object)`, `set`, and `size`. To obtain a formal specifications of the `List` interface, we started from its specification in the Java Modelling Language (JML) that was made available on the JML website [58]. We then built an automatic translation from these JML specifications to FIR. The analyses revealed bugs in some of the implementations, as well as errors in the JML specifications themselves.

6.1.1 Results

The bounded verification analyses were conducted with a scope of 4 list buckets, 4 list values, integers ranging from -8 to 7 (4 bits), and 3 loop unrollings. All experiments were run on a 2.2GHz Intel Pentium 4 machine with 1GB RAM and Ubuntu GNU/Linux 5.10. The complete timing results are shown in Table 6.1.

JDK LinkedList

This experiment analyzed the `java.util.LinkedList` class provided in the Sun JDK. We checked each of the 12 List methods against its JML specification, finding no specification violations. As shown in Table 6.1, no analysis exceeded two minutes.

	add(i,o)	add(o)	clear	contains	get	indexOf	isEmpty	lastIndexOf	remove(i)	remove(o)	set	size
JDK	23.5	16.0	12.6	84.8	16.6	17.2	15.1	62.6	19.8	77.5	18.0	18.7
Trove1.1b5	**18.3**	15.0	12.6	14.6	16.9	20.0	15.6	18.3	22.2	**11.1***	24.0	13.0
Trove0.1.2	**14.2**	13.8	12.8	15.3	**13.0**	19.4	15.6	**15.0**	**13.3**	**20.4**	**13.9**	12.9
ApacheAbstract	20.8	20.2	13.0	64.5	16.4	25.2	20.9	94.9	37.4	34.2	20.9	29.4
ApacheCaching	25.3	24.1	14.9	119.3	23.8	22.9	22.5	84.8	28.5	100.2	24.8	19.1
MITSeeded	**10.1**					**10.7** **10.1**	**10.4**		**11.0**			

Displayed above are the times (in seconds) to check each method against its specification. Each analysis was bounded by 4 list buckets, 4 list values, integers ranging from -8 to 7 (4 bits), and 3 loop unrollings. Specification violations are displayed in **bold**, and the * indicates the violation was intentional and documented by the developer. In the seeded implementations only the seeded methods were checked, and those times are shown collectively in the last row. Two seeded implementations had bugs in `indexOf`.

Table 6.1: Duration of Method Analyses (seconds)

GNU Trove

GNU Trove is a library of collection implementations designed to yield better performance than the Java Collections Framework in special situations. The library includes a class called `TLinkedList`, a linked list implementation that accepts elements implementing an interface that provides four methods: `getNext`, `getPrevious`, `setNext`, and `setPrevious`. `TLinkedList` uses the elements added to it as the actual nodes in the linked list. This is intended to avoid the extra cost of constructing separate node instances. One disadvantage of this design is that it disallows duplicate list entries.

We checked two versions of the `TLinkedList` class for conformance to the JML specifications. The first version was distributed with Trove 1.1b5, which at the time of the study was the most recent version of the library. The second version, version 0.1.2, was released four years earlier and contained a bug in an inner iterator class in `TLinkedList`, according to the project's CVS logs.

Upon checking the most recent version of `TLinkedList`, we found two of its methods to violate the JML specifications, `remove(Object)` and `add(int,Object)`. When the argument to `remove(Object)` is not already contained in the list, the method can behave incorrectly. Upon inspection of the Trove API, we found this behavior to be deliberate; its specification for `remove(Object)` includes a precondition that the element must be contained in the list. We temporarily amended our JML specification to include this precondition and found no further violation of this specification.

The violation in `add(int,Object)`, however, was not deliberate and constitutes a genuine bug in the implementation that was apparently unknown to the developers. The method contains a subtle off-by-one error when inserting into the middle of the list. Checking the older version of `TLinkedList`, we found half of its methods to violate their specifications, including the bug in the iterator's `remove`, but including several other bugs as well.

Apache Commons-Collections

Commons-Collections is a library offered by the Apache Jakarta project that contains collection implementations to supplement those in the standard Sun library. It includes two linked list implementations, both of which we analyzed. The first, `AbstractLinkedList`, is a standard linked list that provides the same functionality as the Sun implementation, except that it is written with finer-grained procedural abstraction, giving potential subclasses more flexible implementation support. Although it is an abstract class, it contains no abstract methods, so we were able to check it directly for its conformance to the JML specifications.

The second class analyzed was `NodeCachingLinkedList`. Like the `TLinkedList` class in GNU Trove, this implementation attempts to mitigate the cost of constructing new nodes on each addition to the list. To do so, it maintains a separate linked list of nodes that have already been constructed and reuses nodes from this cache whenever possible. Nodes are added to the cache upon element removal, but the cache is constrained to not exceed a preset maximum size. We found no specification violations in either implementation.

MIT Seeded Implementations

As an exercise on test coverage, students in an MIT undergraduate course in software engineering were asked to write comprehensive test suites of the Java `List` interface. To measure the coverage of each student's suite, the suites were executed on a series of mutant versions of the Java `LinkedList` implementation. The mutants were each generated by seeding a single bug in one of the `LinkedList` methods, causing the method to violate its specification.

Five of these mutants contained bugs in one of the 12 methods considered in this experiment. We used our tool to check the mutant methods and successfully detected the bug in each one. We did not check the remaining methods, because they directly delegated to the Sun implementation, which we had already determined did not violate the specifications.

6.1.2 Scope Effects

We ran further analyses to determine the smallest scope needed to detect each of the specification violations. No violation required more than a single loop unrolling to be revealed, and all but one violation was detected when linked lists were limited to a length of 2 and integers to a bit width of 3. The remaining violation – the apparently unknown bug in the latest version of the Trove library – required 3 buckets and 4-bit integers for its detection.

To evaluate the scalability of the analysis, we re-analyzed the `add(int, Object)` method in the Sun implementation for progressively larger scopes. When bounded by 5 buckets, 5 values, 4 loop unrollings, and 4 bits to an integer, the analysis completes in about 2 minutes. When increased to 6 buckets, 6 values, and 5 loop unrollings, it takes about 20 minutes. If the bounds are increased to 7 buckets and 7 values and integers are increased to 5 bits, the analysis continues for one hour before timing out.

6.1.3 Specification Errors

In the process of checking these implementations against the published JML specifications, our tool revealed two errors in the specifications themselves. These errors were corrected during the course of the analysis, and the data shown in Table 6.1 used only the corrected specifications.

The first error, found in the specification of the `add(int, Object)` method, was discovered when our tool reported a specification violation in the Sun implementation. The JML specification states that the method adds a specified element at a specified index in the list when the following condition is true:

```
requires 0 <= index && index < this.size();
```

— and that it throws an exception otherwise. However, the Sun specification states that the element should be added even when the index is *equal* to the size, in which case it should be added to the very end of the list.

A second error was found in the `indexOf` method when the tool failed to find a bug in a seeded implementation. The JML specification says, correctly, that if the method does not return -1, then the specified element must be in the list. However, it omits the necessary inverse: if the method *does* return -1, the element must *not* appear in the list.

6.2 Case Study 2: Electronic Voting Software

This study analyzed the code of the KOA Remote Voting System. First deployed for public elections in the Netherlands in 2004, KOA is an open-source, internet voting application written in Java. Intended to be used primarily by expatriots, KOA stands for the Dutch phrase "Kiezen op Afstand" which means "voting at a distance."

The KOA application contains a small vote-tallying subsystem that processes the ballot data and counts the votes. The vote-tallying module was developed independently of the rest of the application by the Security of Systems (SoS) research group at the University of Nijmegen, the developers of ESC/Java2 [23]. In building the module, SoS annotated their Java source code with specifications in the Java Modeling Language and used their own ESC/Java2 tool to check the code against those specifications [49, 34, 52, 51]. The code was also subject to about 8,000 tests from the unit-testing tool *jmlunit* [19].

We applied Forge to check the KOA vote-tallying code against the provided JML specifications. The analysis was limited to eight classes, listed in Table 6.2, that form the core of its functionality. The `AuditLog` class logs the progress of the vote-tallying; `Candidate` records the tally of an individual candidate; `CandidateList` pairs a list of candidates in an election with a `CandidateListMetadata` that stores additional properties of the election; `District`, `KiesKring`, and `KiesLijst` are kinds of political district boundaries; and `VoteSet` records the cumulative tally for all candidates in the election. The *methods* column in the table lists the number of methods analyzed in each class.

class	methods	sloc	slocc	dslocc	dslocc/ method	violations	mean scope	mean time (sec)
AuditLog	90	286	1237	1617	18.0	1	5.0	2.3
CandidateListMetadata	10	72	246	643	64.3	1	5.0	33.6
Candidate	12	103	363	1013	84.4	1	5.0	59.3
KiesKring	15	119	482	1272	84.8	5	5.0	249.7
District	6	53	163	543	90.5	0	5.0	18.5
KiesLijst	12	111	367	1432	119.3	4	5.0	104.6
CandidateList	13	130	493	1746	134.3	3	4.5	1416.8
VoteSet	11	113	400	2688	244.4	4	3.7	1783.9
Sum or *Mean*	**169**	**987**	**3751**	**10954**	*64.8*	**19**	*4.9*	*262.7*

Table 6.2: Summary analysis statistics of each class. Means are calculated over the analyses of the methods within a class, not over successive analyses of the same class.

When Forge is applied to the methods of a class, the performance of the analysis depends not only upon the complexity of the code in the class but also upon the complexity of its specification, as well as the specifications of classes upon which that class depends. The *sloc* column in Table 6.2 gives the number of source lines of code in each class; *slocc* includes code and comment lines. Because JML is written inside Java comments, the slocc measures, albeit indirectly, the complexity of the class' code and specification. The *dslocc* is the *slocc* plus the number of lines of comment in classes upon which the class directly depends. The *dslocc/method* approximates the complexity of a modular analysis of a method within the class.

As shown in the table, we applied Forge to a total of 169 methods of varying complexity across the eight classes. The *violations* column lists the number of methods that were found to violate their specification. A total of 19 specification violations were found. The experiments were run on a Mac Pro with two 3GHz Dual-Core Intel Xeon processors and 4.5GB RAM running Mac OS X 10.4.11. (Forge is single-threaded and so it did not take advantage of the multiple cores.) The code on which these analyses were conducted was, at the time of this study, the latest version available in its Subversion repository.

We initially analyzed each method in a scope of 5 instances of each type, an integer bitwidth of 4 (integers -8 to 7), and 3 loop unrollings. Most analyses completed quickly, but a few of the more complex methods exceeded our timeout of four hours. For those that timed-out, we progressively lowered the scope until the analysis completed within the time limit. The *mean scope* column in the table lists the average maximum scope in which the analysis successfully completed, and *mean time (sec)* is the mean time in seconds for a successful analysis. Note that these means are calculated over the analyses of the methods within a class, not over successive analyses of the same class. As shown in the table, the average analysis time is roughly correlated with the dslocc/method measure.

class	method	error	min bound
CandidateListMetadata	init	under	1 / 3 / 1
KiesKring	addDistrict	bug	1 / 3 / 1
VoteSet	addVote(String)	over	1 / 3 / 1
KiesLijst	clear	over	1 / 3 / 3
AuditLog	getCurrentTimeStamp	over	2 / 1 / 1
Candidate	init	under	2 / 3 / 1
CandidateList	addDistrict	under	2 / 3 / 1
CandidateList	addKiesLijst	over	2 / 3 / 1
CandidateList	init	over	2 / 3 / 1
KiesKring	addKiesLijst	bug	2 / 3 / 1
KiesKring	init	under	2 / 3 / 1
KiesKring	make	under	2 / 3 / 1
KiesLijst	addCandidate	over	2 / 3 / 1
KiesLijst	compareTo	bug	2 / 3 / 1
KiesLijst	make	over	2 / 3 / 1
VoteSet	addVote(int)	over	2 / 3 / 1
VoteSet	validateKiesKringNumber	over	2 / 3 / 1
VoteSet	validateRedundantInfo	over	2 / 3 / 1
KiesKring	clear	over	2 / 3 / 3

Table 6.3: Specification violations: error classification and minimum bound (scope / bitwidth / unrollings) necessary for the error's detection.

6.2.1 Specification Violations

Table 6.2.1 gives statistics on the 19 specification violations detected. The methods named `init` in the table are constructors. To evaluate the "small-scope hypothesis", for each violation found we progressively lowered the bound on the analysis (scope of each type, bitwidth of integers, number of loop unrollings) until the analysis no longer detected the counterexample. The minimum bound under which each counterexample is found is given in the last column of Table 6.2.1.

Every specification violation can be attributed to one of two causes: a bug in the code or an error in the specification. As outside observers, we can make educated guesses as to the cause, but classifying the violation with complete certainty requires knowing the programmer's intention. Does the specification accurately reflect the programmer's intention, in which case the violation indicates a bug in the code; or did the programmer err in transcribing his or her intention into the specification?

Specification errors themselves can be divided into two subcategories: *overspecification* and *underspecification*. A case of *overspecification* occurs when the specification of the method under analysis requires too much from the implementation – either its pre-condition is too weak or its post-condition is too strong. A case of *underspecification* occurs when the method under analysis calls a method whose specification provides too little to the caller – either the pre-condition of the called method is too strong or its post-condition is too weak. (Recall that our analysis, being modular,

assumes that the specifications, not the implementations, of called methods define their behavior.)

Specification errors, while not "bugs" per se, are still important to address. For example, there may be methods whose correctness depends on the overspecification of a called method. Fixing the overspecification may, therefore, reveal latent bugs in dependent methods. In contrast, a case of underspecification doesn't pose an immediate problem, but it could allow a bug to be introduced in the future. That is, the underspecified method's implementation could be changed at a later date in a way that still satisfies its contract, but causes dependent methods to fail.

As shown in Table 6.2.1, of the 19 violations found by Forge, we believe 3 to be due to buggy code, 11 due to overspecification, and 5 due to underspecification. From our follow-up "smallest scope" analyses of each violating method, we found that every violation would have also been found in a scope of 2, a bitwidth of 3, and 3 loop unrollings. In fact, all but two of the violations required only 1 loop unrolling, the exceptions being `KiesKring.clear` and `KiesLijst.clear`. Both `clear` methods contain loops with if-statements in the body of the loop, and 3 unrollings were necessary to cover all paths. Additionally, four violations needed only the minimal scope of 1 and one violation was found in the minimal bitwidth of 1.

A minimal bitwidth of 3 (integers from -4 to 3) was needed for nearly every analysis, because some static array fields in the code were required to be of at least length 2. Lowering the bitwidth to 2 would allow a maximum integer of only 1. These arrays were not required to be fully populated, however – they could contain null elements – so their minimal length requirements did not in turn affect the minimal scope necessary to detect violations.

6.2.2 Example Violations

In this section, we present and discuss a sample of four specification violations detected by our analysis, two of which we've classified as bugs, one overspecification, and one underspecification. For brevity, some of the code excerpts and JML specifications shown below have been simplified.

(a) KiesLijst.compareTo [bug]

The body of this method contains a correct implementation of the `compareTo` method in `Kies`*`Kring`*, not `Kies`*`Lijst`*. This is likely a copy-and-paste error:

```
class KiesLijst {
  public int compareTo(final Object an_object) {
    if (!(an_object instanceof KiesKring)) {
      throw new ClassCastException();
    }
    final KiesKring k = (KiesKring) an_object;
    return number() - k.number();
  }
}
```

The `instanceof` check and the initialization of local variable `k` should refer to `KiesLijst`, not `KiesKring`.

(b) KiesKring.addDistrict [bug]

The `KiesKring` class stores an array of districts in the `my_districts` field and a count of the number of districts in the `my_district_count` field. The specification for `KiesKring` includes an invariant that the count is equal to the number of non-null entries in the array:

```
private final /*@ non_null @*/ District[] my_districts
private byte my_district_count;
//@ invariant my_district_count == (\sum int i; 0 <= i && i < my_districts.length;
//@                                              (my_districts[i] != null) ? 1 : 0);

boolean addDistrict(final /*@ non_null @*/ District a_district) {
  if (hasDistrict(a_district)) {
    return false;
  }
  my_districts[a_district.number()] = a_district;
  my_district_count++;
  return true;
}
```

Each district has a number that is used as its index in the array. The `hasDistrict` method returns true when the `my_districts` array contains a district with the same number and name as its argument. Thus, if the `a_district` argument has the same number but a *different* name than a district already in the array, the method will overwrite an existing district and increment the district count in violation of the invariant. The district count should only be updated only if there is no existing district at that index.

This violation might be classified as a specification error if the programmer forgot an invariant prohibiting two districts from having the same number but different names. The rest of the code does not appear to rely on such an invariant, however. Indeed, the `District.equals` method checks for equality by comparing not only the number but also the name.

(c) VoteSet.addVote [overspecification]

This method suffers from overspecification in the form of a missing precondition. Note that it invokes the method `Candidate.incrementVoteCount`:

```
class VoteSet {

  final void addVote(final int a_candidate_code) throws IllegalArgumentException {
    if (!(my_vote_has_been_initialized && !my_vote_has_been_finalized)) {
      throw new IllegalArgumentException();
    }
    final Candidate candidate = my_candidate_list.getCandidate(a_candidate_code);
    candidate.incrementVoteCount();
    candidate.kiesLijst().incrementVoteCount();
  }
}

class Candidate {

  //@  requires my_vote_count < AuditLog.getDecryptNrOfVotes();
  //@  modifies my_vote_count;
  //@  ensures my_vote_count == \old(my_vote_count + 1);
  int incrementVoteCount() { . . . }
}
```

As shown, `incrementVoteCount` has a precondition that the number of votes for the candidate be less than a preset number, but `addVote` does not ensure this condition. We believe the programmer erred in not including the stronger inequality constraint in the precondition of `addVote`. It is also possible that the programmer intended `addVote` to be robust when the inequality is false, in which case we would re-classify this violation as a bug.

(d) KiesKring.init [underspecification]

The post-condition of the `KiesKring` constructor invokes an underspecified `KiesKring.name` method:

```
//@ requires a_kieskring_name.length() <= KIESKRING_NAME_MAX_LENGTH;
//@ ensures number() == a_kieskring_number;
//@ ensures name().equals(a_kieskring_name);
private /*@ pure @*/ KiesKring(final byte a_kieskring_number,
                               final /*@ non_null @*/ String a_kieskring_name) {
  my_number = a_kieskring_number;
  my_name = a_kieskring_name;
}

//@ ensures \result.length() <= KIESKRING_NAME_MAX_LENGTH;
/*@ pure non_null @*/ String name() { return my_name; }
```

The specification of the constructor claims that calling `name()` in the post-state yields a string equal to the `a_kieskring_name` argument, and the constructor does indeed assign the argument to the `my_name` field. However, even though the implementation of `name` returns the `my_name` field, its specification says merely that it returns some string whose length is less than a fixed constant. Thus, the post-condition of `name` is not strong enough to establish the post-condition of `init`. Indeed, an implementation of `name` that always returns the empty string would satisfy its weak specification, but would clearly cause `init` to violate its own specification.

6.2.3 Why were these problems missed?

Prior to our case study, the KOA software had been the subject of rigorous development. The code, written according to a "verification-centric methodology" [52], had been checked with ESC/Java2 and unit-testing. Despite these prior efforts, Forge found that 19 of the 169 methods it analyzed to violate their specifications.

On one level, it may seem surprising that our bounded verification technique revealed specification violations missed by unit testing. After all, bounded verification is conceptually a form of testing, in which all tests up to some small size are executed. However, two properties of our technique distinguish it from unit testing in key ways. One important difference is that our technique is modular and, therefore, can detect problems due to underspecification of called methods, while unit testing cannot. While underspecification does not indicate the presence of bugs today, it invites bugs in the future as later modifications are made to the code.

The major difference, however, is one of coverage. The voting code was subject to nearly 8,000 unit tests, which on the face of it sounds like a large number of tests

to generate and run. Our technique, however, by exploiting the efficiency of Kodkod and its underlying SAT-solving technology, is capable of analyzing thousands, if not millions, of scenarios of every method individually.

Despite these differences, it still came as a surprise that unit testing did not detect the buggy `KiesLijst.compareTo` method discussed in Section 6.2.2a. Perhaps the static type of the `compareTo` parameter being `Object` (not `KiesLijst`) caused the testing tool to feed the method arguments only of *runtime* type `Object`. In these cases, the buggy version of the method would behave correctly by raising a `ClassCastException`.

To catch the bug in `hasDistrict`, discussed in Section 6.2.2b, a unit-test would need to first populate the pre-state with a non-empty array of districts, and then pair the pre-state with a district argument with the same number but a different name from an existing district in the array. Revealing such a bug requires a higher level of coverage than can be expected from traditional unit-testing.

It is difficult for us to determine why ESC/Java2 failed to detect the specification violations found in our study, and, unfortunately, the authors of the ESC study were unable to provide additional information in this regard. We do know that ESC/Java2 was unable to establish the post-condition on 53% of the KOA methods it checked, so perhaps all 19 violations fell into this category. Or perhaps it did claim to establish the post-condition on some of those 19, but due to unsoundness in ESC, some were in reality faulty. Kiniry, Morkan, and Denby [53] detail the sources of unsoundness and incompleteness in ESC/Java2.

Two examples where the unsoundness of ESC may have played a role are the analysis of KOA methods `KiesKring.clear` and `KiesLijst.clear`. ESC/Java2 examines only one unrolling of each loop, but our case study found that at least three unrollings were necessary to detect the overspecification of those methods.

6.3 Case Study 3: Strategy/Coverage Evaluation

This study uses 10 small benchmark Java methods to compare the performance of the symbolic execution strategies, test the small-scope hypothesis, and evaluate the effectiveness of the coverage metric. First, each of the 8 symbolic execution strategies are run on the benchmarks and their relative performance is evaluated. Next, the 10 benchmark methods are automatically seeded with library of mutations, yielding 429 mutant methods. To test the "small-scope hypothesis," we find the smallest bound that kills each mutant.

These mutants methods are then used as a basis for evaluating the coverage metric. Some of the mutants cannot be killed by bounded verification, because the mutation led to an infinite loop in the code. We apply the coverage metric to these mutants to test its ability to find missed statements in the presence of such non-terminating behavior. Lastly, we evaluate the coverage metric's ability alert the user of an insufficient bound. To do so, we apply bounded verification to mutants, but in a deliberately insufficient bound, and for the mutants that survive, we note the number of missed statements reported by the coverage metric.

name	loc	los	fields	arrays	loops	nested	recurse	alloc	arith
bubble	16	3		×	×	×			×
conloop	8	2	×		×				
conrec	5	2	×				×		
copytree	8	2	×				×	×	
createlist	10	2	×		×			×	
insertion	11	3		×	×	×			×
multiply	15	2			×	×			×
register	11	2	×					×	
search	15	2		×	×				×
treeadd	22	5	×		×			×	

Table 6.4: Characteristics of the benchmark problems.

Here's a short description of each benchmark method:

- *bubble*: bubble sort
- *conloop*: checks whether an element is contained in a list using iteration (a loop)
- *conrec*: checks whether an element is contained in a list using recursion
- *copytree*: copies a binary tree data structure recursively
- *createlist*: creates a linked list of a specified length
- *insertion*: insertion sort
- *multiply*: implements multiplication using addition within nested loops
- *register*: web registration procedure from Chapter 4
- *search*: binary search
- *treeadd*: inserts an element into a binary search tree

Characteristics of the ten benchmark methods are shown in Table 6.4. The table lists the the number of lines of code ("loc") and number of lines of specification ("los") of each method, followed by whether the method does any of the following: accesses fields ("fields"); accesses arrays ("arrays"); contains loops ("loops"); contains nested loops ("nested"); contains recursion ("recurse"); performs dynamic allocation ("alloc"); or performs integer arithmetic ("arith"). The benchmarks were hand-picked to offer a diversity of these features.

Unlike the previous case studies whose specifications were in the Java Modeling Language [58], the specifications of these methods were written in the JForge Specification Language [90] (JFSL), a Java specification language based on relational logic. (The example analysis of the linked list in Chapter 1 also used specifications written in JFSL.) The benchmarks require so few lines of specification due to the concision with which specifications can be expressed in JFSL, at least relative to JML.

All of the experiments in this section were run on a Mac Pro with two 3GHz Dual-Core Intel Xeon processors and 4.5GB RAM running Mac OS X 10.5.6.

6.3.1 Performance of Symbolic Execution Strategies

Recall from Chapter 3 that three kinds of statements — assign, branch, and create — can be symbolically executed with either an inline or constrain rule. These rules can be arbitrarily combined, for a total of 8 strategy combinations. This study begins by empirically comparing the performance of the 8 strategies on the 10 benchmarks.

The experiments confirmed our intuition that the inline strategy would usually lead to better performance than the constrain strategy. Since the inline strategy adding variables and constraints to the relational logic problem, it leads to smaller SAT problems, and as a result faster SAT-solving times. Let's examine the effect in miniature on a small FIR code snippet and specification:

code:	**spec:**
x := a	$x = x_{old}$
x := x ∪ b	

The inline strategy declares only three relations — x_0, a_0, b_0 for the initial values of x, a, and b — and generates one relational logic formula:

$$(a_0 \cup b_0) = x_0$$

If Kodkod takes on the order of n Boolean variables in SAT to represent a relation, then inlining produces a SAT encoding roughly on the order of $3n$ Boolean variables and n Boolean formulas to encode the equality.

The constrain strategy, in contrast, declares five relations — x_0, a_0, b_0, x_1, x_2 — and generates three formulas:

$$x_1 = a_0$$
$$x_2 = x_1 \cup b_0$$
$$x_2 \neq x_0$$

This requires roughly $5n$ Boolean variables and $3n$ Boolean formulas and results in a larger SAT problem and longer solving time.

The chart in Figure 6-1 summarizes the results of the analyses. Along the x-axis are each of the symbolic execution strategies. Each strategy is abbreviated "aA cC bB", where A is the strategy rule used for assignment statements, C is the rule for create statements, and B is the rule for branches, and A, B, and C are each either 'I' for "inline" or 'C' for "constrain." For example, "aI bC cC" is a strategy in which assignment statements are inlined and branch and create statements are constrained. The chart shows the average number of variables and clauses in the resulting SAT problem and the average time of the analysis, across the the 10 benchmark problems, as a factor of those values for the "aI cI bI" strategy. For example, it shows that the "aC cC bC" strategy, compare to "aI cI bI", generated on average a SAT problem with almost 7 times as many variables and about 3 times as many clauses, and took slightly over 3 times as long to solve.

As shown in the chart, using the constrain rule for create statements has only a slight impact on the size of the problem and the speed of the analysis. This is likely due to the sparsity of dynamic allocations in Java code. Using the constrain rule for assignments has a bigger impact: both "aC cI bI" and "aC cC bI" almost double the

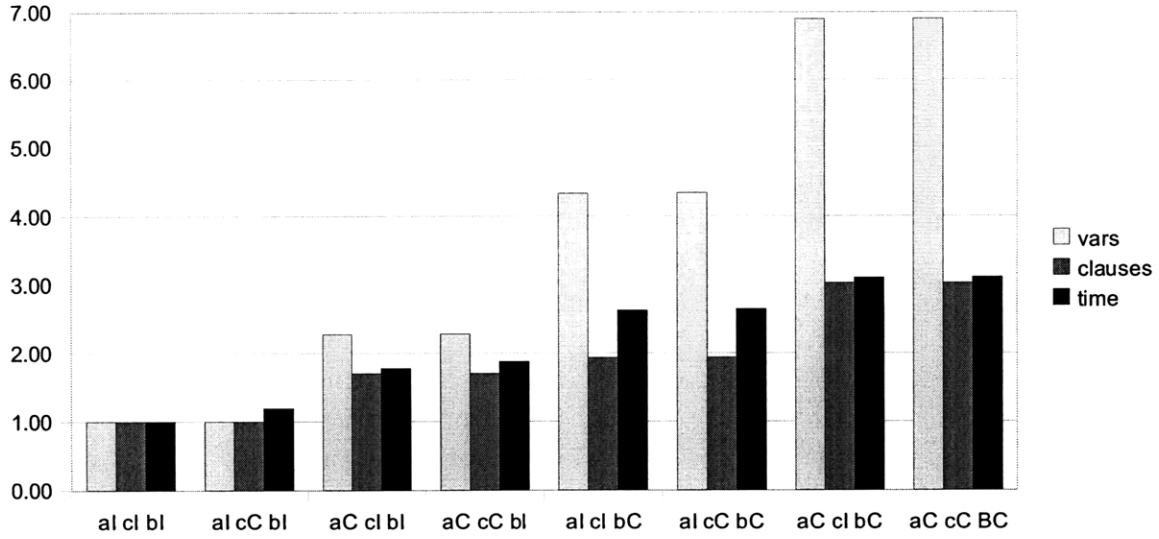


Figure 6-1: Strategy Performance Comparison. The bars show the average number of variables, clauses, and time-to-solve for the SAT problems generated by each strategy, as a factor of those numbers for the "aI cI bI" strategy.

solving time of the analysis. Using the constrain rule for branch statements, though, causes the biggest slow-down. Each time it is applied, it introduces new relations for every FIR variable that is modified by either side of the branch, as well as two additional constraints for each of those variables. Naturally, the biggest formulas and slowest times come from the "aC cC bC" strategy, which uses the constrain strategy for every statement it encounters.

6.3.2 Mutant Generation

The remainder of the study involves the application of Forge to mutant versions of the benchmark problems. To generate the mutants, we used MuJava [61], a tool which automatically applies a library of *mutation operators* to a user-provided method. Each operator in the library, shown in Table 6.5, each makes a single change to the method to yield a new mutant. The Arithmetic Replacement Operator (AOR), for example, replaces an arithmetic operator in the code with another, such as replacing a plus symbol (+) with a minus (-). For another example, the Conditional Operator Insertion (COR) inserts a Boolean negation symbol (!) before a Boolean expression. When applied to the 10 benchmarks methods, the mutant operators generated 429 mutants.

To evaluate the small-scope hypothesis — the hypothesis that bugs will usually have small counterexamples — we applied the bounded verification analysis to each of 429 mutants to find the smallest bound in which each mutant method produced a counterexample, i.e., failed to satisfy its specification. Using standard testing terminology, an analysis that finds a counterexample in a mutant method is said to have *killed* the mutant.

Operator	Description
AOR	Arithmetic Operator Replacement
AOI	Arithmetic Operator Insertion
AOD	Arithmetic Operator Deletion
ROR	Relational Operator Replacement
COR	Conditional Operator Replacement
COI	Conditional Operator Insertion
COD	Conditional Operator Deletion
SOR	Shift Operator Replacement
LOR	Logical Operator Replacement
LOI	Logical Operator Insertion
LOD	Logical Operator Deletion
ASR	Assignment Operator Replacement

Table 6.5: MuJava Mutation Operators

Not every mutant can be killed by our bounded verification analysis, however. This is for two reasons. The first reason is that some mutants are *equivalent mutants*; that is, they behave the same as the original method on all inputs. For example, the following code that computes the absolute value of a number x:

```
if (x < 0) then abs = -x else abs = x
```

is equivalent to the mutant where the less-than (<) operator has been replaced with a less-than-or-equal-to (<=) operator:

```
if (x <= 0) then abs = -x else abs = x
```

Equivalent mutants cannot (and should not) be killed by any analysis, bounded verification or otherwise.

The second reason bounded verification may be incapable of killing a mutant is that the mutation introduced non-termination. This is not to say that bounded verification can never find a counterexample in a method with an infinite loop. Indeed, if the method with an infinite loop has at least one terminating execution on which it behaves incorrectly, then that incorrect, terminating behavior will still be found (in a sufficient bound). But what our bounded verification analysis cannot reveal (regardless of the bound) is non-termination itself, due to its use of finite loop unrolling. For a mutant with an infinite loop to survive bounded verification, it must be equivalent to the original program on all of its terminating inputs.

Table 6.6 shows the number of mutants generated from each benchmark method and their classification into three categories. The "Mutants" column lists the total number of mutants generated from that method. "Equivalent" lists the number of those that behave equivalently to the original on all inputs. "Infinite" lists the mutants that have non-terminating behavior but behave equivalently to the original on terminating inputs. The number of "killable" mutants are those that are neither "equivalent" or "infinite" – they are the ones that can be killed by bounded verification. Of the 429 mutants, 46 were equivalent and 16 infinite, leaving 367 killable.

Benchmark	Mutants	Equivalent	Infinite	Killable	Min Bound
bubble	71	5	4	62	1s 3b 2u
conloop	18	0	0	18	1s 2b 2u
conrec	22	2	0	20	1s 2b 1u
copytree	8	0	0	8	2s 2b 1u
createlist	21	3	4	14	2s 4b 1u
insertion	67	4	6	57	1s 3b 3u
multiply	71	8	2	61	0s 3b 2u
register	6	2	0	4	2s 2b 1u
search	99	17	2	80	1s 4b 3u
treeadd	46	3	0	43	3s 2b 1u
total/*max*	**429**	**46**	**16**	**367**	*3s 4b 3u*

Table 6.6: Mutants per Benchmark and Minimal Bound for Detection of All Killable Mutants. A mutant is not killable by bounded verification if it is equivalent to the original method on all of its terminating executions. A bound of "sS bB uU" is a scope of S on each type, a bitwidth of B, and U loop unrollings

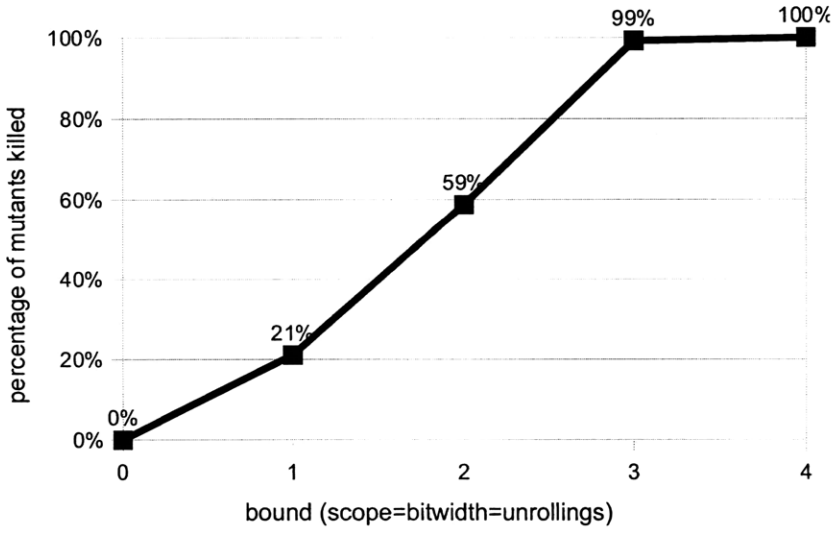


Figure 6-2: Percentage of Mutants Killed per Bound. A bound of n is a scope of n, bitwidth of n, and n loop unrollings.

Benchmark	Mutant	Missed
bubble	original	2
bubble	AOI_S64	+12
bubble	AOI_S66	+12
bubble	AOI_S72	+3
bubble	AOI_S74	+3
createlist	original	1
createlist	AOI_S3	+3
createlist	AOI_S5	+3
createlist	AOR_B1	+3
createlist	AOR_B2	+3

Benchmark	Mutant	Missed
insertion	original	2
insertion	AOI_S80	+12
insertion	AOI_S80	+12
insertion	AOI_S82	+12
insertion	AOI_S88	+5
insertion	AOI_S90	+5
insertion	AOI_S101	+1
insertion	AOI_S103	+1
multiply	original	0
multiply	AOI_S13	+11
multiply	AOI_S19	+2

Table 6.7: Infinite Loop Detection. The "missed" column lists the number of missed statements measured in an infinitely-looping mutant compared to the original method.

The "Min Bound" column of Table 6.6 shows the minimal bound needed to kill all the method's killable mutants. As shown, all 367 killable mutants are killed with a scope of 3, a bitwidth of 4, and 3 loop unrollings. The graph in Figure 6-2 shows the percentage of killable mutants that were killed per bound. In a bound of 1 (scope of 1, bitwidth of 1, 1 loop unrolling) 21% of all killable mutants were killed. In a bound of 2: 59%; bound of 3: 99%; and all were killed in a bound of 4.

6.3.3 Infinite Loop Detection

Although bounded verification cannot detect non-terminating behavior, the coverage metric, as discussed in Chapter 4, can detect statements that the bounded verification missed when non-terminating behavior occurs. In this part of the case study, we apply the coverage metric to the 16 mutants that were unkillable due to non-terminating behavior and compare the number missed statements measured in the mutant to the number of missed statements found in the original method. (There are missed states in the original method due to weakness in the specification or dead or redundant code.) If the coverage metric measures more missed statements in the mutant, then it has the potential to help users detect cases of non-termination.

The results of the 16 infinitely-looping mutants are shown in Figure 6.7. The mutant named "original" refers to the original, non-mutant method and is listed to show the number of missed statements in the original method. The number of missed statements is listed as $+n$ to indicate that the coverage metric found n more statements in the mutant procedure to be missed compared to the original. On all 16, the coverage metric found more missed statements in the mutant than in the original, with the difference in missed statements ranging as high as 12 statements.

Here is the code of mutant `AOI_S80` of the `insertion` method, which performs insertion sort on an array:

```
00  static void insertion(int[] a) {
01    for (int i = 1; --i < a.length; i++) {
02      int j = i;
03      int B = a[i];
04      while (j > 0 && a[j - 1] > B) {
05        a[j] = a[j - 1];
06        j--;
07      }
08      a[j] = B;
09    }
10  }
```

The mutation is in Line 01, where a decrement operator (`--`) was inserted before the variable `i` in the loop condition (`--i < a.length`). This decrement offsets the impact of the increment (`i++`), causing the loop to continue infinitely. The 12 statements in the FIR procedure that the coverage metric finds to be missed correspond to the body of the for-loop. If the body of the for-loop were removed, this mutant will still have no counterexamples within any finite number of unrollings.

6.3.4 Insufficient Bound Detection

This final part of the case study evaluates the coverage metric's ability to help the user detect an insufficient bound on the analysis. It asks the question, "If the bound on the analysis is smaller than needed for a bug's detection, does the coverage metric measure more missed statements?" If the metric does measure more missed statements when the bound is insufficient, it can potentially alert users to the insufficiency.

The evaluation was conducted as follows. For each benchmark method, we looked up the minimal bound, found from the prior small-scope analysis, needed to detect all killable mutants of that method. If the minimal bound was "sS bB uU" — scope of S, bitwidth of B, and U loop unrollings — we analyzed all the mutants of the method in three separate bounds: "s($S-1$) bB uU" (a scope one smaller than necessary to reveal the bug in every mutant), 'sS b($B-1$) uU" (a bitwidth one smaller than necessary), and 'sS bB u($U-1$)" (one fewer loop unrolling than necessary). Some mutants were still killed in these smaller bounds, but many survived. For those that survived, we recorded the number of missed statements the coverage metric reported and compared it to the number of missed statements of the original method in the original bound. If the coverage metric reported more missed statements on the mutant, we recorded the insufficient bound as "detected."

The results of these analyses are shown in Table 6.8. For each of the insufficient bounds, the table lists the number of killable mutants that "survived" in that bound, followed by the number of those mutants in which the insufficient bound was "detected" (the coverage metric found more missed statements in the mutant). A dash (—) in the table means the the bound already at their minimal level and, therefore, could not be decremented. As shown in the table, all 260 cases in which a mutant survived in a decremented scope, the coverage metric was able to detect the insuf-

benchmark	scope - 1		bitwidth - 1		unrollings - 1	
	survived	detected	survived	detected	survived	detected
bubble	62	62	56	56	19	19
conloop	18	18	1	0	2	2
conrec	20	20	1	1	—	—
copytree	6	6	2	0	—	—
createlist	7	7	1	0	—	—
insertion	57	57	54	54	3	1
multiply	—	—	6	2	2	2
register	4	4	1	0	—	—
search	79	79	2	0	1	0
treeadd	7	7	10	5	—	—
total	**260**	**260**	**135**	**118**	**27**	**24**

Table 6.8: Insufficient Bound Detection.

ficient bound. It was less successful at detecting an insufficiency in the bitwidth or number of loop unrollings, but still found 118 out of 135 decremented bitwidths and 24 out of 27 decremented unrollings. In total, the metric was able to detect 402, or 95%, of the 422 cases of an insufficient bound.

6.4 Chapter Summary

These case studies accomplished several goals. First, they showed Forge to be effective at finding previously unknown bugs in code. In the linked list case study, a previously unknown bug was discovered in the latest version of the GNU Trove library. In the voting study, Forge found 19 of the 169 methods it analyzed to violate their specifications. In the final study, all of the non-equivalent mutants were either killed or found by the coverage metric to have more missed statements than the original.

Second, the studies provided additional support for the small-scope hypothesis. In all three studies, every bug was found in a very small bound. Increasing the scope from 2 to 5 in the voting study and 3 to 6 in the linked list study revealed no additional errors. In the final benchmark study, all of the killable mutants were killed in a scope of 3, bitwidth of 4, and 3 loop unrollings.

The benchmark study addressed additional questions the prior studies did not. It provided empirical evidence to support our intuition that the inline strategy would produce smaller SAT problems, and therefore faster solving times. The benchmark study also substantiated the effectiveness of the coverage metric. The metric was capable of measuring more missed statements in all infinitely-looping mutants and in 95% of the cases where the bound was too small.

Since the symbolic execution strategy that inlines whenever possible generally leads to the best performance, we have configured it to be the default strategy in Forge. However, the coverage metric, which depends upon the constrain strategy, proved to be effective for the detection of infinite loops and insufficient bounds, so Forge switches to the constrain strategy when the metric is enabled.

Chapter 7

Discussion

This chapter puts Forge in the context of related work, reflects on its usefulness, and discusses how it might be improved. The chapter begins by comparing Forge to related efforts to verify code with relational logic. It then divides the remaining related work into three key categories: languages similar to the Forge Intermediate Representation; program analyses related to our bounded verification technique; and coverage metric similar to ours. Finally, drawing on insights from the related work, the results of the case studies, and general experience with the Forge framework, the chapter includes some final thoughts on the tool and suggestions for future directions.

7.1 Verification with Relational Logic

There have been three very related efforts to perform bounded verification via a translation to relational logic: Vaziri's Jalloy tool [85], Taghdiri's specification-inference techniques [79], and Galeotti's verification with DynAlloy [39].

7.1.1 Jalloy

A predecessor to Forge, Vaziri's Jalloy tool was the first effort to perform bounded verification via a translation to relational logic. Jalloy checked Java methods by translating them to the Alloy modelling language [48] and solving the resulting Alloy model with the Alloy Analyzer. Like its underlying Alloy analysis, Jalloy placed a scope on each type and a bitwidth on the integers, and to completely finitize the method under analysis, Jalloy limited the number of loop unrollings. Jalloy also inlined the bodies of all called methods, so its analysis was not modular.

Vaziri demonstrated the feasibility of Jalloy on individual methods and data structures, but not on real programs composed on multiple interacting components. Forge builds on the ideas of Jalloy, but exploits new model-finding technology and introduces new features, including a new translation to logic, an intermediate representation with specification statements, and a coverage metric, and it incorporates these features into a framework for program analysis that is applicable to real programs.

Where Jalloy used the Alloy Analyzer 3.0 as its model finder, Forge exploits the latest advances offered by the Kodkod model finder. These advances include improved symmetry breaking, improved sharing detection, and support for partial instances [84]. Some of these new features, like symmetry breaking, require no special work on the part of Forge to take advantage of: they are provided by Kodkod automatically. Others, however, like partial instances, Forge must make an effort to exploit. Partial instances are the constants that can be provided as a lower bound on a relational logic variable (Section 3.1.1). One place where they are exploited is in fixing a total ordering over each domain to optimize the symbolic execution's treatment of dynamic allocation (Section 3.4.2).

Unlike Forge, Jalloy did not use a symbolic execution to translate code to relational logic. Instead, it encoded the *computation graph* of a procedure in an Alloy model [85]. Compared to Jalloy's encoding, Forge's symbolic execution is arguably easier to understand, because it emulates the actual execution of the code. It also lends itself to an inductive proof of its correctness (Section 3.3). Although not a symbolic execution, Jalloy's translation was similar to the constrain strategy, because it declared new relations to store the values of modified variables. As evidenced by the third case study (Section 6.3), the constrain strategy tends to not perform as well as the inline strategy. Dolby and Vaziri have since updated the Jalloy translation with a logarithmic encoding of integers [32], an idea worth exploring in Forge.

Jalloy translated Java to relational logic directly, with no intermediate representation in between. One benefit to using an intermediate representation is a reduced burden to building analyses for other high-level languages beyond Java. Toshiba Research, for instance, is actively developing CForge [75], a front-end that translates C code to FIR. FIR also introduced a clear separation of concerns into the design of the framework; it separated the task of choosing a relational representation for program constructs (performed by JForge and CForge) from the task of encoding imperative code in a declarative formula (performed by Forge). Most importantly, FIR's inclusion of specification statements supports a refinement methodology and allows analyses of high-level programs to be modular. JForge, for example, abstracts away the implementation of called methods into statements of their specification. Jalloy, in contrast, can only inline the body of called methods, a key factor that limits its scalability.

7.1.2 Karun

Taghdiri developed techniques for *specification inference* [79] and integrated them into an early version of JForge. These techniques allowed JForge to analyze methods without requiring specifications be written of called methods, nor requiring the full body of those called methods be inlined. Instead, Taghdiri's analysis, embodied in a tool called Karun, infers partial specifications of the called methods from their implementations.

Karun begins by performing an abstract interpretation of each called method to obtain an initial, conservative abstraction of its behavior [80]. It then replaces each method call with its abstraction, obtaining an abstracted version of the origi-

nal method. From there, it follows a standard CEGAR (Counter-Example Guided Abstraction Refinement) approach. Karun applies Forge's bounded verification on the abstracted method. If the abstracted method satisfies the specification, then the original method will necessarily satisfy it, so Karun terminates.

If, on the other hand, bounded verification finds a counterexample, the counterexample witnesses a pre-/post-state pair for each method call. For each of these method calls, Karun invokes the bounded verification again to search for an execution of the called method that conforms to the pre-/post-state pair previously witnessed. If there exists such an execution, then the counterexample was valid, and Karun reports the counterexample to the user.

If no such execution exists, the counterexample is invalid and the abstraction is refined. To refine the abstraction, Karun queries the underlying Kodkod model finder for an unsatisfiable core of the analysis, which includes formulas generated from the called method's implementation that prohibit the existence of the execution. The formulas in the core are then conjoined to the abstraction of the method, and the analysis of the method is performed anew. This process continues until no counterexamples are found (the method is correct) or a valid counterexample is found (the method is incorrect). In the worst case, the method calls are eventually refined to the entire behavior of the called methods, the equivalent of inlining the method call.

Taghdiri found Karun to scale better than inlining when the specification being checked was a partial property of the method's behavior. In these cases, usually only a limited specification for each called method needs to be inferred, and the analysis completes more quickly than it would have had the method calls been inlined. When checking a method against a very strong specification, however, the full behavior of called methods is often necessary, and Karun winds up effectively inlining the behavior of the called methods anyway. In these latter cases, the analysis would have performed better by inlining the called methods upfront rather than waiting for Karun's CEGAR process to complete.

Karun's CEGAR approach would be worth incorporating into the current version of Forge framework. Doing so would offer Forge the option of performing more like a standard static analysis, meaning a non-modular analysis of code with respect to a partial property of its behavior. Being non-modular, the results of Karun's analysis would not hold if the implementation of a called method were to change, and it would not scale to strong properties, but some users may prefer that tradeoff to the work required to write specifications of called methods. Importantly, incorporating Karun would allow Forge to refrain from taking a position on this tradeoff: it could support both a modular/strong-property mode and a non-modular/weak-property mode. Users could use either or both, and in different combinations, as they see fit.

Support for both modes of analysis would make Forge more conducive to an *incremental* style of specification. A user could begin with a partial specification of a method and no specifications of called methods, and yet still apply Karun's CEGAR approach to check the method against the partial property. But as the user's desire for greater confidence increases, the specification can incrementally become more detailed and as Karun's analysis slows, the user can add specifications of called methods and transition to a modular analysis.

7.1.3 Verification with DynAlloy

DynAlloy [38] is an extension to the Alloy modelling language that includes actions from dynamic logic. Very recently, Galeotti, et. al. [39] have begun exploring an alternative bounded verification approach that involves translating Java code to DynAlloy, which is in turn translated to Alloy and analyzed with the Alloy Analyzer 4.0. Their initial results demonstrate significant performance gains from adding symmetry-breaking predicates on fields that are known in advance to be acyclic. This optimization would be worth exploring in our own work.

7.2 Related Languages & Representations

The two areas of work related to the Forge Intermediate Representation include: (1) work on on relational programming languages and (2) work on intermediate representations for program verification.

7.2.1 Relational Programming Languages

FIR is not the first programming language based on sets and relations. An early example is SETL [30], a high-level programming language founded on set theory and set operations. SETL and FIR have differences in terms of expressiveness. While in FIR relations may be of any arity, in SETL they can be at most binary. SETL, on the other hand, can expression higher-order sets — sets that contains sets — whereas all sets and relations in FIR must be first-order. If relations in FIR could be higher-order, it would not be amenable to analysis with the Kodkod model finder, which allows only first-order relations. Like FIR, SETL offers set operators and universal and existential quantification, but not being designed for the purposes of verification, offers no form of specification statement.

Unlike SETL and like FIR, the CrocoPat Relational Manipulation Language [12] allows relations of any arity. Unlike FIR, CrocoPat does not support dynamic object allocation. Instead, their operations are performed on a finite pre-defined universe of objects. CrocoPat does not offer any notion of a specification statement.

7.2.2 Intermediate Representations

Due to the complexities of dealing with high-level programming languages, many program-verification techniques encode high-level programs in an intermediate representation (IR) that is more amenable to analysis [59]. ESC/Java [37] and ESC/Java2 [23] encode Java in a variant of Dijkstra's guarded commands [31]; Boogie [9] encodes .NET bytecode in BoogiePL [25]; the Bogor model checker [72] encodes Java in the Bandera Intermediate Representation [46]; and Krakatoa [62] and Caduceus [36] encode Java and C, respectively, into the Why language [35]. These intermediate representations facilitate transformations (e.g. loop-unrolling and call-inlining) and optimizations, and they simplify the eventual translation to verification conditions.

Most of these languages, including ESC's IR, BoogiePL, and BIR, are variants and extensions of Dijkstra's guarded command language (GCL). One difference between these GCL-based representations and FIR is in their support for branches in the control-flow graph. While FIR offers a standard if-statement (a Boolean condition and two successor statements), the GCL languages offer a more general "selection" statement in which multiple guards may be enabled at once, and the control flow passes non-deterministically to one of the enabled guarded commands.

Since high-level code contains only standard if-statements, we were not motivated to have a more general construct in FIR. Putting selection statements in FIR would have also complicated the inline symbolic execution rule for branches, because the conditional expressions that the rule constructs relies on the strict binary choice offered by if-statements. A lack of selection statements is not a fundamental limitation on FIR's expressiveness, because any degree of non-determinism can be obtained with additional specification statements. For example, in the following code snippet, control from the if-statement flows non-deterministically to statement S_T or S_F, based on the result of a non-deterministic assignment to the Boolean variable b:

```
b := spec(true)
if b then S_T else S_F
```

In contrast to FIR, none of these IRs encode the heap of a program as relations. Instead, they use a mix of arrays, uninterpreted functions, and axioms in first-order logic. Since transitive closure cannot be axiomatized in first-order logic [45], some properties expressible in FIR, including common reachability properties of data structures, cannot be expressed in these IRs. For example, in FIR one can refer to all the nodes reachable from the root of a tree with the expression root.*(left + right), where * is reflexive, transitive closure. With first-order logic alone, the closest alternative is an inductively defined reachability function on nodes that encodes only an abstraction of reachability [60]. Because this abstraction permits more states than are possible, an analysis that uses it risks producing false alarms.

A related language that *is* founded on relations is DynAlloy [38], an extension to the Alloy modelling language that includes actions from dynamic logic. While DynAlloy was originally designed as a usable modelling language, not an IR, it has been used as an intermediate language for automated test-case generation [40] (discussed below) and, more recently, for bounded verification [39] (discussed above).

7.3 Related Program Analyses

The Forge bounded verification analysis shares similarities with work in testing, theorem proving, model checking, and shape analysis. These will be compared to Forge primarily along four dimensions: (1) modularity — several of these approaches, e.g. model checking, conduct whole program (non-modular) analyses; (2) automation — some, e.g. theorem proving, are not fully automated; (3) property strength - many target partial properties, not strong specifications; and (4) coverage — some, e.g. testing, do not offer high degrees of coverage of (and therefore confidence in) the program's behavior, while others provide greater coverage in the form of a proof.

7.3.1 Testing

Testing shares with our bounded verification analysis an ability to check strong functional correctness properties in a fully automated way. One key difference from our approach, however, is that testing is a whole-program, not a modular, analysis. That is, testing does not assume the specifications of called procedures; it executes them directly. The benefit of a whole program analysis is that once it is complete, one can be confident, without any further investigation, that the procedure under analysis will yield the correct results for the inputs on which it was tested. A modular analysis of the same procedure, in contrast, requires a subsequent analysis of the called procedures to check that they satisfy their specifications. On the other hand, if the implementation of a called procedure changes, even if it continues to satisfy its specification, there is no guarantee from testing that the analyzed procedure (the caller) will still behave correctly on the tested inputs.

Compared to testing, one disadvantage of Forge is that it can only analyze code that can be translated into the intermediate representation. Currently, this means Forge cannot be applied to code that performs real-number arithmetic, interacts with I/O devices, or spawns new threads, as we have not yet developed techniques to translate these into FIR. While Forge cannot analyze components that use these features directly, it can still analyze methods that depend on components that do, provided that specifications for the components can be written to abstract those features from the method under analysis. This is currently the case, for example, with JForge's handling of Java `HashMap`. The implementation of `HashMap` performs real number arithmetic to calculate load factors on its internal hash table, but an analysis of a method that uses `HashMap` is shielded from those details, seeing it only as a relation that maps keys to values.

Another difference with our approach is that testing requires, with some exceptions, that the specification (or oracle) be written imperatively in code. While this lowers the learning curve, it also makes some kinds of specifications a challenge to write. For instance, writing post-conditions that refer to the pre-state requires manually introducing auxiliary variables that store their values, a laborious task if the procedure under analysis updates fields across many objects, e.g. a procedure that balances a binary search tree. Writing frame conditions or calculating a transitive closure can be tedious, as well. It's also difficult to express specifications of abstract data types and their abstraction functions imperatively. That would require somehow executing the abstraction function, which might not be feasible.

Testing and bounded verification also differ in terms of coverage. Testing usually exercises only a tiny fraction of the input space, while our bounded verification approach can quickly explore the equivalent of millions or billions of tests. While there are coverage metrics for testing, these metrics measure the degree to which code constructs (e.g. statements, branch, paths) are exercised, but they do not measure to what extent the input space of the procedure (e.g. procedure arguments and state of the heap) are explored. Bounded verification, in contrast, covers the entire input space within a small bound.

Exhaustive Testing

To address the lack of coverage in traditional testing, some recent work has explored "exhaustive testing," an approach in which all tests within a bound are executed. Implemented in the tools TestEra [63] and Korat [15], exhaustive testing generates every test case that meets a user-provided precondition. With TestEra, the precondition is provided as a formula in the Alloy modelling language [48] and the test inputs are generated with the Alloy Analyzer. In Korat, the precondition is a Java predicate (a method that returns Boolean) and the inputs are generated with backtracking search.

Exhaustive testing offers some advantages over our approach. When applied to the analysis of an individual method that does not depend on other components, exhaustive testing may very well scale better than our approach. Especially when the precondition is highly constrained (few valid inputs), separating the generation of those few inputs from their execution will likely perform better than combining the input and execution constraints into one large formula. As all forms of testing, it also faces fewer problems from programs involving real numbers and I/O, that cannot yet be analyzed by Forge. (Currently, Forge reports and error if these features are encountered and will not proceed with the analysis.)

When analyzing code that depends on several components, however, exhaustive testing, being non-modular and unable to treat those components abstractly, may find less success. Particularly when the precondition is relatively unconstrained, it can become prohibitively expensive to generate all the test-cases within a small bound. When TestEra was applied to the Galileo fault tree analysis tool [78], for instance, it initially consumed too much memory to generate all test cases within a scope of 3, a scope in which there were 1,276,324 fault trees. Successfully generating these (fairly unconstrained) inputs required user-written post-processing scripts to offload aspects of the generation and domain-specific symmetry-breaking predicates. In analyses of code involving several dependent components, such as the voting system from Section 6.2, or where the inputs are relatively unconstrained, bounded verification is likely to offer better performance.

Unlike both Forge and TestEra, Korat accepts specifications written as Java predicates, not relational logic formulas, a feature with both pros and cons. On the one hand, allowing specifications in Java lowers the learning curve for a user of the tool. On the other hand, as discussed above, it makes certain specifications inconvenient to express. Writing the representation invariant of a binary search tree (BST), for example, requires a carefully crafted worklist algorithm that is longer and arguably more error-prone than the concise definition of BST possible in a relational logic [15].

Java Automated Testing with Dynalloy

Another approach to increasing the coverage offered by testing is offered by Galeotti and Frias's Java Automated Testing (JAT) tool [40]. By translating Java to DynAlloy, an extension of the Alloy modelling language to include actions from dynamic logic, JAT is capable of generating test suites that match user-chosen coverage criteria, including statement, branch or path coverage. Due to its use of Alloy, JAT relies upon the same finite model-finding technology and SAT solving used by our approach.

For each test case that is required by the criteria, JAT constructs a formula in DynAlloy that is true if such a test case exists. If path coverage is chosen, for example, JAT enumerates every path through the code and constructs a formula that is true if the execution follows that path. Either that path is infeasible (within the bound provided to the analysis), or its solution corresponds to a test case that takes that path. So instead of automatically generating all tests within a bound, like exhaustive testing, JAT generates just those tests needed to meet a user-chosen coverage metric. Unlike bounded verification and exhaustive testing, this approach does not guarantee any measurable coverage of the heap shapes, which could limit its ability to find bugs.

7.3.2 Theorem Proving

The basic process followed by an automated theorem prover is very similar to our own. From the code of a procedure, they both generate logic formulas — "verification conditions" — that are then discharged by one or more decision procedures. In our analysis, that decision procedure is Kodkod, which requires a bound to finitize the search space and make the problem decidable. In theorem proving, the decision procedures do not assume a bound, but as a result leave many problems undecidable. These approaches rely upon additional user feedback, such as user-provided loop invariants, to complete the proof.

There are two common methods that theorem provers use for generating verification conditions. The traditional approach is to apply a *weakest precondition calculus* [31] that pushes the specification backwards through the code of a procedure to determine the weakest precondition on the procedure that guarantees it will satisfy the specification. It is not uncommon, however, for theorem provers to perform a forward traversal of the code using symbolic execution [69], as we do. One advantage of using symbolic execution is that the specification is needed only at the end of the process, not in advance. As a result, symbolic execution, unlike weakest preconditions, allows the same procedure to be checked against several specifications without traversing the entire procedure anew for each specification.

While some theorem provers operate only on simple input representations, others have been integrated into frameworks for verifying software directly. One example is Boogie [9], which verifies programs written in Spec#, an extension of C# that includes a strong specification language. Boogie translates Spec# code and specifications into the BoogiePL intermediate representation and then encodes BoogiePL in verification conditions accepted by the Z3 SMT solver. Another framework is Jahob [56], which verifies Java code using a combination of different decision procedures.

Theorem proving techniques require a lot of effort and expertise on behalf of the user. The decision procedures they employ are often not powerful enough to complete the desired proof, thus requiring additional input and reasoning from the user. The cost and difficulty of this effort often make theorem proving impractical. For example, it is common for a theorem prover to carry a learning curve of about 6 months, even for a highly skilled software developer [87, 6].

ESC/Java2

The Extended Static Checker for Java (ESC/Java) [37] and its successor ESC/-Java2 [23] bring automation to, and thereby increase the practicality of, theorem proving techniques, but they do so by the sacrificing soundness and completeness in their analyses. The ESC tools analyze code by generating verification conditions in a predicate first-order logic and discharging those conditions with the Simplify theorem prover [29]. While ESC/Java checked only weak specification properties such as the absence of null pointer dereferences, ESC/Java2 targets strong specifications expressed in the Java Modelling Language [58].

To make its analysis fully automated, ESC/Java2 makes a number of compromises that render its analysis neither sound nor complete [53]. Like bounded verification, it unwinds loops, and therefore does not require user-provided loop invariants. It also reasons about integers as if they are unbounded, i.e., not limited by a bitwidth as they truly are in Java. The Simplify theorem prover upon which ESC/Java2 depends also has its own limitations. Since it may loop forever, Simplify bounds its analysis time in advance. It also relies upon a heuristic pattern-driven instantiation of universal quantifiers to guide its proof search, but in the presence of arbitrary first-order logic specifications provided by the user, the reliance on pattern matching can cause Simplify to miss seemingly "obvious" proofs. These limitations may explain why the bugs and specification errors we detected in the voting system case study (Section 6.2) were missed by ESC/Java2.

7.3.3 Model Checking

Model checking was originally designed as a technique for verifying concurrent, finite-state transition systems against properties written in temporal logic. These characteristics made early model checkers well-suited to checking the conformance of hardware systems to control-intensive specifications [22]. Extending model-checking techniques to software was inhibited by at least two factors. First was the need to encode high-level programming languages into the input languages of the model checkers, such as the Promela language analyzed by SPIN [43]. This problem was addressed in part by the Bandera toolset [24], which translates Java code into the languages of several model checkers and maps their counterexamples back to Java source code. Second, the complexity of realistic software systems exacerbated the *state-space explosion* that arises from trying to reason about all feasible execution paths, a challenge which limited the application of early model checkers to relatively small systems.

Abstraction-Based Model Checking

The model checkers SLAM [8], BLAST [13], and MAGIC [18], mitigate state-space explosion through the the use of counterexample guided abstraction refinement (CEGAR). These model checkers map the potentially infinite states of a software system into a finite number of abstract states. A property is checked by searching all the reachable states of the abstract machine. If a counterexample is found, the model

checker employs a separate decision procedure to determine whether it is spurious, a possibility due to the abstraction.

In the standard CEGAR approach, a spurious counterexample causes the abstraction to be refined, the result of which is a new abstract transition system, and the model checking is then performed anew. This refinement process continues until a non-spurious counterexample is found or no counterexamples are detected. This process may continue infinitely, causing the model checker not to terminate. It may also produce false alarms due to limitations in the decision procedures that determine whether a counterexample is spurious.

Because these model checkers rely on abstraction, the specifications they check cannot be strong enough to require the full, precise behavior of the code. They seem to be limited to checking very partial, temporal safety properties.

SAT-Based Model Checking

Another approach to controlling state-space explosion is by bounding the length of execution paths examined. This approach, known as *bounded model checking* and employed by the model checkers BMC [14] and NuSMV [21], unrolls the transition relation a finite number of times, encodes the question of whether the resulting system satisfies the provided property in a Boolean satisfiability (SAT) problem, and invokes a SAT solver. The bounding technique and reliance on SAT make this quite similar to our approach.

While these similarities make it tempting to label Forge a kind of bounded model checking, the term "model checking" brings certain connotations to bear that would not apply to our work. The term "model checking" connotes that the specifications being analyzed are expressed in a temporal logic. It also suggests that the tool performs a non-modular, whole-program analysis that might involve standard model checking techniques like partial-order reductions or state hashing. None of these are true of Forge. Instead, we see Forge as being a bit closer to a formal verification approach in that it generates verification conditions and then discharges them with a decision procedure — the only difference being that the decision procedure we use is bounded. Perhaps a useful analogy: bounded model checking is to traditional model checking as bounded verification is to traditional verification.

Saturn [89] is another program analysis technique that encodes the behavior of a procedure in SAT. It handles loops by either finite unwinding (which as in our approach can cause errors to be missed), or by assuming they can cause arbitrary state changes (which can generate spurious errors). Saturn checks whole programs against properties expressible as finite state machines, a weaker specification language that that supported by our approach. It has so far been used successfully to detect two kinds of errors: (1) invalid locking sequences in Linux and (2) memory leaks. Saturn generates small summaries that abstracts the behavior of called procedures, an approach which allows it to scale, but which is another source of spurious errors.

Model Checking Strong Specifications

All the model-checking approaches discussed thus far have been limited in terms of the complexity of the code and the strength of the specifications they can analyze. However, a few recent model checkers, namely Bogor [72] and Java PathFinder [86], can both handle the full or nearly full complexity of high-level programming languages and can check programs in those languages against strong specifications. They are both explicit-state model checkers that perform whole-program analyses.

To our knowledge, Bogor is the first application of traditional model-checking techniques to the analysis of strong specifications. It translates Java code to the Bandera Intermediate Representation (BIR), a guarded command language with support for object-oriented features, including dynamic object allocation, threads, garbage collection, and dynamic dispatch. The specifications it checks are a large subset of the Java Modelling Language [58]. Bogor uses symmetry and partial-order reductions to mitigate the state-space explosion introduced by these features. While there have been demonstrations of its efficiency on small examples [72], being a whole-program (non-modular) analysis, it is unclear whether it scales to real programs.

Like Bogor, Java PathFinder (JPF) can check strong specifications, though it is not a "traditional" model checker. Unlike most model checkers, JPF does not explore every path of a model, but instead dynamically executes Java bytecode itself and reports runtime assertion failures. Because it executes arbitrary runtime assertions, JPF can check strong properties, but writing specifications in Java faces all the challenges discussed earlier in the context of testing (Section 7.3.1). To control state-space explosion, JPF makes unsound approximations, including state hashing, a technique which treats unequal states as the same if they hash to the same value. Moonwalker [73] is an adaptation of the JPF approach to check .NET bytecode.

A limitation of both Bogor and JPF is that they only analyze *closed systems*. That is, they must be given a fixed pre-state they then attempt to analyze all behaviors originating from that state, but they do not analyze code over all contexts of use. As a result, they are often combined with test harnesses that automatically generate inputs and repeatedly execute the tool on each. This strategy makes achieving high code coverage challenging, particularly with the analysis of library code which could be called in a wide variety of contexts. And if a program invokes outside modules whose code is not available, then the program cannot be analyzed at all.

To adapt the techniques of Bogor and JPF to the analysis of *open systems*, the developers of both have turned to symbolic execution. Out of the work on JPF has come JPF-SE [4] and from Bogor has come Kiasan [26], both of which use techniques of their predecessors instead treat their inputs as symbolic values drawn from unbounded domains and perform a symbolic execution of the code starting with those values. Using symbolic values eliminates the need for a test harness and permits modular analysis that does not require the implementation of called procedures. Due to looping in programs, symbolic execution by itself may not terminate. To address this problem, JPF-SE puts a finite limit on its search depth and the length of the symbolic formulas it generates. More similar to our approach, Kiasan limits the size of linked data structures, and as a last resort, will limit the number of loop unrollings.

7.3.4 Shape Analysis

The goal of shape analysis is the verification of programs with respect to data structure properties. While these properties include deep, structural invariants on the heap, they are nevertheless weaker than the arbitrary first-order logic specifications targeted by our approach. But whereas our approach verifies programs within bounded heaps, the goal of shape analysis is proving the correctness of the code for heaps of unbounded size. Because the goal is a proof, shape analyses must either sacrifice full automation (and require that addition of user-provided annotations like loop invariants) or completeness (and allow false alarms).

The Three-Valued Logic Analyzer implements a technique called *parametric shape analysis* (PSA) [74], which conservatively approximates the potential heap configurations at each program point with a *shape graph*. The shape graph encodes the points-to relationships of the heap using a three-valued logic that includes, *true*, *false*, and an *unknown* value to represent uncertainty. TVLA requires the user provide predicates for the abstraction and an operational semantics that defines how a set of primitive actions affect the abstract state. Given an input shape graph, it performs an abstract interpretation of the program and iterates until the abstract state reaches a fixed point, reporting an error if it violates the user-provided property. Like other techniques based on abstract interpretation, TVLA cannot produce a counterexample for rejected programs, and it may reject correct ones. It has been used to verify quite complex properties, including that a list-reversal procedure is correct.

The pointer assertion logic engine (PALE) [64] is another approach to shape analysis. PALE encodes the program and its specification in a monadic second-order logic formula, whose validity is then checked with the MONA tool [54]. Because it does not perform abstract interpretation, PALE does not require user-provided predicates and does not issue false alarms like TVLA. Also unlike TVLA, it reports counterexamples for any program it rejects. However, it is capable of analyzing only loop-free code and, thus, requires the user annotate all loops with invariants in advance. MONA has a worst-case non-elementary time complexity (bounded by a stack of exponentials whose height is proportional to the formula length), but has been shown to be efficient in practice.

7.4 Related Coverage Metrics

While coverage metrics have existed for decades in testing, most static analyses lack anything comparable. The one exception is in the area of model checking, where recent work on coverage metrics is what inspired our development of a coverage metric for Forge. This section compares our coverage metric to those designed for testing and model checking, and it concludes by pondering whether a coverage metric could be developed for static analyses based on decision procedures, like ESC/Java.

7.4.1 Coverage Metrics for Testing

The Forge coverage metric has the same purpose as coverage metrics for testing: to identify pieces of the code that were not exercised by the analysis, information a user may in turn use to make the analysis more comprehensive. Of the many coverage criteria for test suites, the Forge coverage metric is closest to statement coverage. The statement coverage of a test suite are the statements that were exercised by the suite, and the Forge coverage metric, to the first order, indicates which statements were exercised by the bounded verification analysis.

Nevertheless, there are at least two key differences between statement coverage in testing and the Forge coverage metric. First, the Forge metric labels statements as covered if they were found to be necessary *for correctness*, while the statement coverage of a test suite includes every statement executed, regardless of whether the success of a test depended on it. Second, our coverage metric, unlike testing metrics, can also indicate whether the specification itself was covered by the analysis.

7.4.2 Coverage Metrics for Model Checking

Coverage metrics for model checking originated from work on *vacuity detection* [57]. A specification is *vacuously satisfied* by a model if replacing a subformula of the specification with an arbitrary formula does not affect its validity. For example, if the specification is an implication whose antecedent is always false for the given model, then the specification is vacuously satisfied, because the consequent can be replaced with any formula to no effect. Vacuity can occur due to an accidental overconstraint in the model or underconstraint in the specification, directly analogous to the cases of overcontrained code or underconstrained specification detected by our technique. Unlike our approach, the technique for determining vacuity does not depend upon an unsatisfiable core detected by their analysis, but by re-running the model checker to search for a witness of the model that satisfies the specification non-vacuously.

This work on vacuity detection spurred the development of coverage metrics for model checking by Chockler [20]. Among her proposed metrics was one for *code coverage*, which defined coverage in the same way as this work: a statement is not covered if its absence does not affect the satisfaction of the specification. She did not experiment with the metric, but suggested it as one among many possible instantiations of a more general coverage algorithm for model checking. Our work can be seen as some empirical validation of the effectiveness of her code coverage definition, at least within the context of our SAT-based technique.

7.4.3 A Coverage Metric for ESC?

Despite the advances in coverage metrics for model checking, static analyses based on decision procedures, including ESC/Java2, provide no comparable measure. The importance of knowing that a seemingly successful analysis may have "gone wrong" is not lost on users of ESC/Java2. To quote its developers [53], "User awareness of the soundness and completeness of the tool is vitally important in the verification

process, and lack of information about such is one of the most requested features from ESC/Java2 users." The closest feature in ESC/Java2 is the automated warning it produces when the tool uses unsound or incomplete reasoning. When analyzing code with a loop, for example, this warning feature notifies the user that "ESC/Java2 does not consider all possible execution paths through a loop." Such automated warnings might be a useful addition to Forge in addition to the coverage metric.

Perhaps ESC/Java2 and other techniques based on automated theorem proving could incorporate a coverage metric like that presented in this thesis. Like our approach, these techniques could maintain a mapping from each statement to the verification conditions generated from that statement. Then, if the automated theorem prover could log the verification conditions necessary for the proof, those condition could be mapped back to "covered" statements in the code.

7.5 Conclusions

The results of the case studies suggest the Forge framework could effectively reduce the number of bugs in code if integrated more broadly into software development practice. Prior to the electronic voting case study, the KOA software had been the subject of rigorous development. The code, written according to a "verification-centric methodology" [52], had been analyzed with both ESC/Java2 and unit-testing. Despite these prior efforts, Forge found 19 of the 169 methods it analyzed to violate their specifications. Similarly, in the linked list case study, a previously unknown bug was discovered in the latest version of the GNU Trove library, and several bugs were detected in an earlier version of the library. In the final study, all of the non-equivalent mutants were either killed by bounded verification or found by the coverage metric to have more missed statements than the original.

The case studies also provide additional support for the small-scope hypothesis. In all three studies, every bug was found in a very small bound. That fact alone, however, is not direct support for the "small-scope hypothesis" since it is possible that the software analyzed is riddled with bugs beyond the bounds chosen. Ideally, one would increase the scope step by step until all errors are revealed, and then determine their distribution. This is possible when the errors are known in advance, as they were in the third study, in which all 367 killable mutants were killed within a scope of 3, bitwidth of 4, and 3 loop unrollings. It was not possible for the other case studies, but we were able to approximate this in miniature by increasing the scope until the analysis became intractable, and by noting the smallest scope in which each error is revealed. Increasing the scope from 2 to 5 in the electronic voting study and 3 to 6 in the linked list study revealed no additional errors, evidence that, at least within that range, the small-scope hypothesis holds.

Although our bounded verification analysis is unsound, as the final case study attests, the coverage metric we developed is capable of mitigating that unsoundness. In all of the mutants that bounded verification failed to kill due to their non-termination, the coverage metric was able to highlight missed statements that could have help the user detect and eventually eliminate the bug. Furthermore, in 95% of the analyses

in which the bound was too small to kill a mutant, the metric was again able to find "suspiciously" missed statements that could have alerted the user to the insufficiency.

Some Future Work. Despite these positive results, we still have concerns with the practicality of our analysis and see several opportunities for future work. One open question, for example, is how much effort is truly required to write formal, declarative specifications of code. In the first two studies, specifications in the Java Modeling Language (JML) had already been written for us, and they were at times quite verbose. Fortunately, initial experience with the JForge Specification Language developed by Yessenov [90], suggests the burden of formal specification is less when the specification language is based on relational logic. Also, when the property analyzed is more partial, incorporating Taghdiri's techniques for specification inference [79], as discussed in Section 7.1.2, could eliminate the need to write specifications of called methods, while still allowing the analysis to scale adequately.

The scalability of the analysis has room for improvement. For the linked list study, the analysis scaled adequately for the task at hand, in all cases terminating within a few minutes. But in the voting case study, while the analysis scaled adequately for most methods, we were disappointed it did not scale to our desired scope of 5 for all of them. One opportunity for improved performance could come from exploiting the presence of generics in Java source code. For example, each Java `Map` in the code, and there are many in KOA, introduces a ternary relation in Kodkod whose second and third columns range over the universe of all objects, an expensive relation to encode in SAT. If the translation from Java to FIR could exploit the type arguments for the keys and values of a `Map`, it could significantly reduce the number of bits needed to represent that ternary relation in SAT and improve the scalability of the analysis.

Looking at related work offers additional ideas for improved performance. Dolby and Vaziri's logarithmic encoding of integers [32] or Galeotti's symmetry breaking for acyclic fields [39], for example, if integrated into Forge could reduce the search space that the Kodkod model finder must explore. Another idea, in use by Kiasan, is to check each path through the code individually, rather than the Forge approach that checks the entirety of the procedure at once. Checking the paths individually opens up an opportunity for parallelization that could improve the performance of the analysis on multi-core machines.

We also see longer-term opportunities for improving the soundness of the analysis by combining our bounded verification technique with decision procedures common to other program analyses. Several modern program verifiers like Boogie [9] use SMT solvers capable of handling multiple decidable theories, a choice that requires them to either limit the richness of their specification language or require additional help from the user. It might be possible to use SMT for just the decidable verification conditions and discharge the undecidable ones with Kodkod.

A Final Thought. This thesis began with the hypothesis that there exists a large, under-served market of developers wishing to check their code against strong, functional-correctness properties. For the developer who can afford to invest very

little in correctness, it seems unlikely that any technique within budget could offer better confidence than testing; and for those who can afford a very high degree of investment, no technique could offer better confidence than theorem proving. What's traditionally been lacking, however, are techniques that offer a good return for the wide swath of developers who can afford a *moderate* degree of investment.

To bridge this gap, there has emerged in recent years a new crop of techniques that aim for a different cost/confidence tradeoff. These techniques, which include ESC/Java2, Java PathFinder-SE, and Kiasan, as well as Forge, seem to defy classification. They don't fit the traditional notion of "testing," a term typically reserved for non-modular, dynamic analyses, and these techniques are neither. At the same time, they are unsound, and therefore don't qualify as "formal verification," either. Perhaps oxymorons like "unsound verification" or "formal testing" are good descriptors. Regardless of their name, we expect and encourage more interest in these techniques in the future. As they have begun to show, high confidence does not always require high cost.

Bibliography

[1] The Astrée static analyzer. http://www.astree.ens.fr/.

[2] FindBugs. http://findbugs.sourceforge.net/.

[3] GNU Trove: high performance collections for Java. http://trove4j.sourceforge.net/.

[4] Saswat Anand, Corina S. Pasareanu, and Willem Visser. JPF-SE: A symbolic execution extension to Java PathFinder. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2007)*, pages 134–138, Braga, Portugal, March 2007.

[5] Alexandr Andoni, Dumitru Daniliuc, Sarfraz Khurshid, and Darko Marinov. Evaluating the "Small-Scope Hypothesis". Technical report, MIT CSAIL, 2002.

[6] Arvind. Why formal verification remains on the fringes of commercial development, May 2008. Formal Methods 2008 Tutorial.

[7] Ralph J. Back. *On the Correctness of Refinement Steps in Program Development.* PhD thesis, University of Helsinki, 1978.

[8] Thomas Ball and Sriram K. Rajamani. The SLAM Project: Debugging system software via static analysis. In *POPL '02: Proceedings of the 29th ACM Symposium on the Principles of Programming Languages*, New York, NY, USA, 2002. ACM Press.

[9] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects (FMCO'05)*, pages 364–387, Amsterdam, The Netherlands, November 2005.

[10] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Proceedings of the Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS'04)*, Marseille, France, 2004.

[11] Boris Beizer. *Software Testing Techniques.* International Thomson Computer Press, 1990.

[12] Dick Beyer. Relational programming with CrocoPat. In *Proceedings of the 28th International Conference on Software Engineering*, May 2006.

[13] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker BLAST: Applications to software engineering. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(5-6):505–525, 2007.

[14] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99), Amsterdam, The Netherlands*, 1999.

[15] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on Java predicates. In *ISSTA '02: Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, July 2002.

[16] C.A.R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4), 1972.

[17] Thomas Ball Carlos Pacheco, Shuvendu K. Lahiri. Finding errors in .NET with feedback-directed random testing. In *International Symposium on Software Testing and Analysis (ISSTA'08)*, Seattle, Washington, July 2008.

[18] Sagar Chaki, Edmund Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 385–395, Washington, DC, USA, 2003.

[19] Yoonsik Cheon and Gary T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In *ECOOP'02*, pages 231–255, Malaga, Spain, June 2002.

[20] Hana Chockler, Orna Kupferman, and Moshe Vardi. Coverage metrics for formal verification. In *Correct Hardware Design and Verification Methods (CHARME)*, October 2003.

[21] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A new symbolic model verifier. In *11th International Conference on Computer Aided Verification (CAV'99), Trento, Italy*, pages 495–499, July, 2003.

[22] Edmund M. Clarke, Anubhav Gupta, Himanshu Jain, and Helmut Veith. Model checking: Back and forth between hardware and software. In *Verified Software: Theories, Tools, Experiments (VSTTE'05)*, volume 4171 of *Lecture Notes in Computer Science*, pages 251–255, Zurich, Switzerland, October 2005. Springer.

[23] David R. Cok and Joseph Kiniry. ESC/Java2: Uniting ESC/Java and JML. In *Proceedings of the Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS'04)*, pages 108–128, Marseille, France, March 2004.

[24] James C. Corbett, Matthew B. Dwyer, John Hatcliff, and Robby. Bandera: A source-level interface for model checking Java programs. *Software Engineering, International Conference on*, 0:762, 2000.

[25] Robert DeLine and K. Rustan M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft, March 2005.

[26] Xianghua Deng, Jooyong Lee, and Robby. Bogor/Kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, pages 157–166, Washington, DC, USA, 2006. IEEE Computer Society.

[27] Greg Dennis, Felix Sheng-Ho Chang, and Daniel Jackson. Modular verification of code with SAT. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'06)*, Portland, Maine, July 2006.

[28] Greg Dennis, Kuat Yessenov Chang, and Daniel Jackson. Bounded verification of voting software. In *Second IFIP Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE'08)*, Toronto, Canada, October 2008.

[29] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, Systems Research Center, HP Laboratories Palo Alto, July 2003.

[30] Robert Dewer. *The SETL Programming Language*, 1979.

[31] E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall, Englewood Cliffs, NJ, 1976.

[32] Julian Dolby, Mandana Vaziri, and Frank Tip. Finding bugs efficiently with a SAT solver. In *ACM/SIGSOFT Symposium on the Foundations of Software Engineering (FSE'07)*, September 2007.

[33] Jonathan Edwards, Daniel Jackson, Emina Torlak, and Vincent Yeung. Subtypes for constraint decomposition. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'04)*, Boston, MA, July 2004.

[34] Fintan Fairmichael. Full verification of the KOA tally system, March 2005. University College Dublin. Bachelor's Thesis.

[35] Jean-Christophe Filliâtre. Why: a multi-language multi-prover verification tool. Technical Report 1366, LRI, Universite Paris Sud, 2003.

[36] Jean-Christophe Filliâtre and Claude Marché. Multi-prover verification of C programs. In *Formal Methods and Software Engineering, 6th International Conference on Formal Engineering Methods (ICFEM'04)*, volume 3308 of *Lecture Notes in Computer Science*, pages 15–29, Seattle, WA, USA, November 2004. Springer.

[37] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02)*, pages 234–245, 2002.

[38] Marcelo F. Frias, Juan P. Galeotti, Carlos G. López Pombo, and Nazareno M. Aguirre. DynAlloy: upgrading alloy with actions. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 442–451. ACM, 2005.

[39] Juan P. Galeotti, Nicolas Rosner, Carlos G. Lopez Pombo, and Marcelo F. Frias. Efficient SAT-based analysis of annotated OO programs by automated computation of tight bounds for class fields. In *Submitted for publication*, March 2009.

[40] Juan Pablo Galeotti and Marcelo Frias. DynAlloy as a formal method for the analysis of Java programs. In K. Sacha, editor, *Software Engineering Techniques: Design for Quality*, volume 227, pages 249–260. IFIP International Federation for Information Processing, Springer, 2006.

[41] John V. Guttag and James J. Horning. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, 1993.

[42] Brian Hackett and Radu Rugina. Region-based shape analysis with tracked locations. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'05)*, pages 310–323, New York, NY, USA, 2005. ACM.

[43] Gerard J. Holzmann. The model checker SPIN. In *IEEE Trans. Softw. Eng.*, volume 23, 1997.

[44] International Atomic Energy Agency (IAEA). Investigation of an accidental exposure of radiotherapy patients in Panama: Report of a team of experts, 26 May – 1 June 2001, August 2001.

[45] Neil Immerman, Alex Rabinovich, Tom Reps, Mooly Sagiv, and Greta Yorsh. The boundary between decidability and undecidability for transitive-closure logics. In *18th International Workshop of Computer Science Logic (CSL'04)*, volume 3210 of *Lecture Notes in Computer Science*, pages 160–174. Springer-Verlag, September 2004.

[46] Radu Iosif, Matthew B. Dwyer, and John Hatcliff. Translating Java for multiple model checkers: The Bandera back-end. *Form. Methods Syst. Des.*, 26(2):137–180, 2005.

[47] Daniel Jackson. Object models as heap invariants. *Essays on Programming Methodology*, pages 247–268, 2000.

[48] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis.* MIT Press, Cambridge, MA, 2006.

[49] Bart Jacobs. Counting votes with formal methods. In *Algebraic Methodology and Software Technology, 10th International Conference (AMAST'04)*, pages 21–22, Stirling, Scotland, UK, July 2004.

[50] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[51] Joeseph Kiniry. Formally counting electronic votes (but still only trusting paper). In *12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007)*, pages 261–269, Washington, DC, 2007. IEEE Computer Society.

[52] Joseph Kiniry, Alan Morkan, Dermot Cochran, Fintan Fairmichael, Patrice Chalin, Martijn Oostdijk, and Engelbert Hubbers. The KOA remote voting system: A summary of work to date. In *Proceedings of Trustworthy Global Computing (TGC'06)*, Lucca, Italy, 2006.

[53] Joseph Kiniry, Alan Morkan, and Barry Denby. Soundness and completeness warnings in ESC/Java2. In *SAVCBS '06: Proceedings of the 2006 conference on Specification and verification of component-based systems*, pages 19–24, New York, NY, USA, 2006. ACM.

[54] Nils Klarlund and Anders Møller. *MONA Version 1.4 User Manual.* BRICS, Department of Computer Science, Aarhus University, January 2001. Notes Series NS-01-1. Available from `http://www.brics.dk/mona/`. Revision of BRICS NS-98-3.

[55] Viktor Kuncak and Daniel Jackson. Relational analysis of algebraic datatypes. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 207–216, New York, NY, USA, 2005. ACM.

[56] Viktor Kuncak and Martin Rinard. An overview of the Jahob analysis system: Project Goals and Current Status. In *NSF Next Generation Software Workshop*, 2006.

[57] Orna Kupferman and Moshe Vardi. Vacuity detection in temporal model checking. In *Correct Hardware Design and Verification Methods (CHARME)*, September 1999.

[58] Gary Leavens. Java Modeling Language. http://www.jmlspecs.org.

[59] Hermann Lehner and Peter Müller. Formal translation of bytecode into BoogiePL. *Electron. Notes Theor. Comput. Sci.*, 190(1):35–50, 2007.

[60] Tal Lev-Ami, Neil Immerman, Thomas W. Reps, Shmuel Sagiv, S. Srivastava, and Greta Yorsh. Simulating reachability using first-order logic with applications to verification of linked data structures. In *20th International Conference on Automated Deduction (CADE'05)*, volume 3632 of *Lecture Notes in Computer Science*, pages 99–115, Tallinn, Estonia, July 2005. Springer.

[61] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. MuJava: a mutation system for Java. In *ICSE '06: Proceedings of the 28th International Conference on Software Engineering*, pages 827–830, New York, NY, USA, 2006. ACM.

[62] C. Marche, C. Paulin-Mohring, and X. Urbain. The Krakatoa tool for certification of Java/JavaCard programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1-2):89–106, 2004.

[63] D. Marinov and S. Khurshid. TestEra: A novel framework for automated testing of Java programs. In *ASE '2001: 16th IEEE International Conference on Automated Software Engineering*, pages 22–31, 2001.

[64] Anders Møller and Michael I. Schwartzbach. The pointer assertion logic engine. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'01)*, June 2001. Also in SIGPLAN Notices 36(5) (May 2001).

[65] Carroll Morgan. The specification statement. In *ACM Trans. Program. Lang. Syst.*, volume 10, pages 403–419, New York, NY, USA, 1988. ACM Press.

[66] Carroll Morgan. *Programming from Specifications.* Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1994.

[67] Joseph M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):287–306, 1987.

[68] National Institute of Standards and Technology. The economic impacts of inadequate infrastructure for software testing, May 2002. http://www.nist.gov/public_affairs/releases/n02-10.htm.

[69] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. *ACM SIGPLAN Notices*, 33(5):333–344, 1998.

[70] Greg Nelson. A generalization of Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(4):517–561, 1989.

[71] David Patterson, Aaron Brown, Pete Broadwell, George Candea, Mike Chen, James Cutler, Patricia Enriquez, Armando Fox, Emre Kiciman, Matthew Merzbacher, David Oppenheimer, Naveen Sastry, William Tetzlaff, Jonathan

Traupman, and Noah Treuhaft. Recovery oriented computing (ROC): Motivation, definition, techniques, and case studies. Technical Report UCB//CSD-02-1175, U.C. Berkeley, March 2002.

[72] Robby, Edwin Rodríguez, Matthew B. Dwyer, and John Hatcliff. Checking strong specications using an extensible software model checking framework. In *Proceedings of the Tenth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *Lecture Notes in Computer Science*, Barcelona, Spain, March 2004. Springer.

[73] Theo C. Ruys and Niels H. M. Aan de Brugh. MMC: the Mono Model Checker. *Electron. Notes Theor. Comput. Sci.*, 190(1):149–160, 2007.

[74] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 105–118, New York, NY, USA, 1999. ACM.

[75] Masahiro Sakai and Takeo Imai. CForge: A bounded verifier for the C language. In *The 11th Programming and Programming Language workshop (PPL'09)*, 2009.

[76] Donna Scott. Assessing the costs of application downtime. Technical report, Gartner Group, May 1998.

[77] Ilya Shlyakhter. *Declarative Symbolic Pure Logic Model Checking.* PhD thesis, Massachusetts Institute of Technology, Februrary 2005.

[78] Kevin Sullivan, Jinlin Yang, David Coppit, Sarfraz Khurshid, and Daniel Jackson. Software assurance by bounded exhaustive testing. In *International Symposium on Software Testing and Analysis (ISSTA'04)*, pages 133–142. ACM, July 2004.

[79] Mana Taghdiri. *Automating Modular Program Verification by Refining Specifications.* PhD thesis, Massachusetts Institute of Technology, February 2008.

[80] Mana Taghdiri, Robert Seater, and Daniel Jackson. Lightweight extraction of syntactic specifications. In *Proceedings of the 14th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE'06)*, Portland, Oregon, November 2006.

[81] Robert Tarjan. Testing flow graph reducibility. In *Proceedings of the fifth annual ACM symposium on Theory of computing (STOC'73)*, pages 96–107, New York, NY, USA, 1973. ACM.

[82] Emina Torlak. *A Constraint Solver for Software Engineering.* PhD thesis, Massachusetts Institute of Technology, 2009.

[83] Emina Torlak, Felix Sheng-Ho Chang, and Daniel Jackson. Finding minimal unsatisfiable cores of declarative specifications. In *FM '08: Proceedings of the 15th international symposium on Formal Methods*, pages 326–341, Berlin, Heidelberg, 2008. Springer-Verlag.

[84] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In *Proceedings of 13th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'07)*, Braga, Portugal, March 2007.

[85] Mandana Vaziri. *Finding Bugs in Software with a Constraint Solver*. PhD thesis, MIT, February 2004.

[86] Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model checking programs. In *ASE '2000: 15th IEEE International Conference on Automated Software Engineering*, pages 3–11, 2000.

[87] Charles B. Weinstock and Fred B. Scheneider. Dependable software technology exchange. Technical Report CMU/SEI-93-SR-004, Software Engineering Institute, Carnegie Mellon University, June 1993.

[88] J. Woodcock and J. Davies. *Using Z—Specification, Refinement, and Proof*. Series in Computer Science. Prentice Hall International, 1996.

[89] Yichen Xie and Alex Aiken. Saturn: A SAT-based tool for bug detection. In *17th International Conference on Computer Aided Verification (CAV 2005), Edinburgh, Scotland, UK*, 2005.

[90] Kuat Yessenov. *The JForge Specification Langauage Reference Manual*, September 2008. http://sdg.csail.mit.edu/forge/refman.pdf.