Automating Modular Program Verification by Refining Specifications

by

Mana Taghdiri

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2008

Author . . . .
Department of Electrical Engineering and Computer Science
December 2007

Certified by . . . .
Daniel N. Jackson
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

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

Automating Modular Program Verification by Refining Specifications

by

Mana Taghdiri

Submitted to the Department of Electrical Engineering and Computer Science
on December 2007, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Modular analyses of software systems rely on the specifications of the analyzed modules. In many analysis techniques (e.g. ESC/Java), the specifications have to be provided by users. This puts a considerable burden on users and thus limits the applicability of such techniques. To avoid this problem, some modular analysis techniques automatically extract module summaries that capture specific aspects of the modules' behaviors. However, such summaries are only useful in checking a restricted class of properties.

We describe a static modular analysis that automatically extracts procedure specifications in order to check heap-manipulating programs against rich data structure properties. Extracted specifications are context-dependent; their precision depends on both the property being checked, and the calling context in which they are used. Starting from a rough over-approximation of the behavior of each call site, our analysis computes an abstraction of the procedure being analyzed and checks it against the property. Specifications are further refined, as needed, in response to spurious counterexamples. The analysis terminates when either the property has been validated (with respect to a finite domain), or a non-spurious counterexample has been found.

Furthermore, we describe a lightweight static technique to extract specifications of heap-manipulating procedures. These specifications neither are context-dependent, nor require any domain finitizations. They summarize the general behavior of procedures in terms of their effect on program state. They bound the values of all variables and fields in the post-state of the procedure by relational expressions in terms of their values in the pre-state. The analysis maintains both upper and lower bounds so that in some cases an exact result can be obtained.

Thesis Supervisor: Daniel N. Jackson
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to thank my advisor Professor Daniel Jackson not only for his invaluable technical guidance on this project over the past few years, but also for his patience, encouragement, and enthusiasm that made the task of preparing talks and papers enjoyable. I also thank my thesis committee members Professor Martin Rinard and Professor Srini Devadas for their helpful comments.

I thank my friends at MIT, Emina Torlak, Robert Seater, Gregory Dennis, Felix Chang, Jonathan Edwards, Derek Rayside, Sarfraz Khurshid, Ilya Shlyakhter, Manu Sridharan, Tina Nolte, Viktor Kuncak, Alexandru Salcianu, and Mandana Vaziri, who made my graduate studies a wonderful academic and social experience. I specially thank Emina Torlak for her insights on the Alloy modeling language, and her implementation of Kodkod, the most efficient Alloy model finder currently available. I thank Gregory Dennis for many useful discussions about checking Java code, and for providing me with his tool Forge. I thank Mandana Vaziri for providing me with Jalloy, a Java bug finder that made it possible for me to evaluate my ideas at the very early stages. I thank Robert Seater, Viktor Kuncak, and Alexandru Salcianu for their valuable comments on my procedure summarization technique.

But above all, I would like to thank my family: my parents Afsaneh and Hossein, and my sisters Shiva and Sara, who always encouraged me to learn and supported me with love throughout my life, and my husband Mehdi, for all his love, patience, encouragement, and support without which this thesis would have never been possible. This thesis is dedicated to my family.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As software systems grow in size and complexity, using conventional techniques such as testing to validate their behavior becomes prohibitively expensive: a large number of carefully designed test cases is required to uncover their errors. Thus developing easy-to-use automatic techniques to increase programmers' confidence in their developed software systems becomes a pressing issue.

Static program verification is one of the main approaches to reasoning about the behavior of a program, in which a program is analyzed using only the information available from the text of the program; no actual execution of the code is involved. Therefore, the code can be analyzed conservatively so that the obtained results account for *all* executions of the program within the analyzed domain.

Static program verification techniques check if a given program conforms to an intended property. Ideally, the property can be a specification describing the complete functionality of the program: what exactly the code is supposed to do. However, since checking an arbitrary piece of code with respect to an arbitrary specification is an undecidable problem, each program verification technique makes a trade-off between the expressiveness of the specification language it handles and its level of automation. Type checkers, for example, are fully automatic but they only check a very restricted class of properties, namely whether the use of variables and procedures conform to their declarations. On the other hand, theorem provers can handle very rich specification languages, but they require guidance from the users.

This thesis introduces a novel program verification technique that automatically checks programs against rich data structure properties. A *data structure property* constrains the structure of the objects in the heap of a program. Checking such properties is particularly important because erroneous manipulations of data structures may cause loss and/or unauthorized access of data, and may eventually cause a wrong behavior in the program. Since data structures stored in the heap can get arbitrarily complex, it can be very hard to check their validity by traditional testing techniques.

The analysis technique described in this thesis aims at minimizing the human cost: it does not require users to provide any annotations beyond the property to check. A procedure is checked against a property using the specifications of its called procedures as surrogates for their code. The specifications, however, are inferred automatically from the code rather than being provided by the user[1]. They are inferred iteratively as needed to check the given property. Therefore, their precision is context-dependent: they encode only as much information about their corresponding procedures as is necessary to check the top-level procedure against the given property.

The thesis also introduces a lightweight static technique that automatically extracts specifications for the procedures of a program. These specifications are not context-dependent: they summarize the general behavior of the procedures in terms of their effect on program state. The technique can handle heap-manipulating code accessing complex data structures. Although used as part of our program verification technique in this thesis, this specification extraction technique is a stand-alone analysis that can be used in a variety of settings.

1.1 Problem Description

1.1.1 Automating Modular Program Verification

Checking data structure properties is a challenging problem. It requires a reasoning engine capable of handling aliasing (whether different pointers point to the same

[1] The specifications that we infer play the role of summaries in traditional analyses, but we prefer to call them specifications to emphasize their partial nature.

object in the heap), and reachability (whether an object is reachable from another object by traversing the fields). It also requires handling different notions that may arise in different kinds of data structures. For example, properties of a 'set' data structure typically involve *membership* and the *cardinality* of the set, a 'map' data structure requires the notion of a *relationship* between values and their keys, and a 'list' structure involves an *ordering* of elements.

Data structure properties arise in heap-manipulating programs: programs whose state can be represented as a graph of objects linked by fields. These programs typically include dynamic features like dynamic object allocation and dynamic dispatch which make them particularly hard to analyze statically.

Due to the complexity of heap-manipulating programs and their behavioral properties, monolithic approaches, where the code is treated as a whole regardless of its modularization, are not practical: they cannot handle large enough programs. Therefore, traditional program verification approaches made extensive use of program structure in structuring the reasoning. Each procedure would be checked against its specification, using the specifications of its called procedures as surrogates for their code. This analysis, called *modular program verification* [28], is usually performed bottom-up in the call tree using the assume-guarantee framework: assuming that the called procedures satisfy their specifications, it is checked if the analyzed procedure guarantees its specification. This approach is attractive because (1) the modularization of the code provides natural abstraction boundaries that can be exploited in reasoning about the behavior of a program, and (2) each procedure is analyzed exactly once, thus providing better scalability.

Automating modular approaches involves solving two problems: (1) given the specifications of all called procedures, how to check a top-level procedure automatically, and (2) how to extract the specifications of the called procedures automatically from the code. One of the main approaches to solving the first problem, i.e. to automating the checking, is to translate the code along with the analyzed property to a set of logical constraints and present it for proof or refutation to a decision procedure. This approach forms the basis of several tools that verify data structure properties of code (e.g. [5,14,21,35,37,58]). None of them, however, automates the specification extraction phase; they all assume that the specifications of called procedures are provided by the user. Boogie [5], for example, extracts verification conditions from a procedure, and presents them for proof to a theorem prover. The tool has been incorporated into the Spec# programming system [6] and applied successfully to substantial programs. However, Boogie expects the user to provide specifications for all procedures that are called directly or indirectly by the analyzed procedure, which can become a considerable burden on the user. This thesis is built on the ideas of Jalloy [58], a counterexample detector for Java programs. Jalloy translates the code and the analyzed property to a boolean formula and solves it using a SAT solver. It inlines all procedure calls whose specifications are not provided by the user. The experiments, however, showed that such inlining does not scale and it is the main obstacle to using Jalloy for checking large systems.

This thesis introduces a strategy to overcome this obstacle. A modular analysis (similar to Jalloy's) is performed that is based on constraint solving and requires specifications of called procedures, but the specifications are inferred from the code rather than being provided by the user. Although in many cases, the inferred specifications are not so compact as they could be if provided by the user, they are small enough to be analyzed automatically.

Of course extracting a specification that summarizes the full observable behavior of a procedure as a logical constraint is not feasible in general. However, in order to check a procedure against a property, it is sufficient to capture only those aspects of the behavior of its callees that are relevant in the context of the calling procedure. Our inference scheme exploits this. In fact, the inferred specifications are sensitive not only to the calling context, but also to the property being checked. As a result, a very partial specification is sometimes sufficient, because even though it barely captures the behavior of the called procedure, it nevertheless captures enough to verify the caller with respect to the property of interest.

Since the specifications are extracted automatically from the code of the called procedures, they are guaranteed to be satisfied by those procedures. Therefore, if a called procedure contains a bug that prevents the top-level procedure from satisfying the analyzed property, that bug will be carried in the inferred specifications and thus, will be uncovered by the analysis.

The analysis technique introduced in this thesis supports a rich specification language – a relational first order logic with transitive closure – sufficient for expressing rich data structure properties. The technique exploits *partiality* of properties. It provides an effective analysis for checking partial properties of code. That is, rather than specifying the full functionality of a program, the analyzed property constrains only a portion of the program's behavior. This is motivated by the observation that the full behavior of a large software system is generally very expensive to check in one analysis. Instead, it can often be specified as several partial properties of the system that can be checked individually, allowing a much more scalable analysis.

In summary, the introduced analysis aims at supporting all the following features.

- **Full Automation.** We provide a fully automatic technique that does not require any user intervention to guide the analysis.

- **No Annotations.** In order to provide an easy-to-use technique, we do not rely on any user-provided annotations, except the top-level property to check.

- **Scalability.** We provide an analysis that can potentially scale to large pieces of code.

- **Modularity.** Although our goal is not to rely on user-provided annotations, our technique can benefit from such annotations whenever available. That is, a user can optimize the analysis of a program by providing specifications for some procedures. This becomes handy in analyzing code with calls to some procedures (e.g. from a library) whose source code is not available.

- **Data Structure Properties.** In spite of all the progress made in checking control-intensive properties of programs, only a few techniques are available to check rich data structure properties. We focus on analyzing properties that express (partial) pre- and post-conditions for some procedure by constraining the configuration of the heap before and after the execution of that procedure.

1.1.2 Existing Approaches

Several static program verification techniques have been developed that support different subsets of the above features. However, to our knowledge, none of them provides a scalable analysis for checking the kind of rich data structure properties that we consider, without requiring extensive user-provided annotations.

SLAM [4] and BLAST [26], for example, provide scalable static analysis for C programs without requiring user intervention or additional annotations. Their scalability is a result of applying a counterexample-guided abstraction refinement framework [11] that computes incremental abstractions of the program that are much simpler to check than the original code. Because the abstractions are computed iteratively, only as needed, the parts of the program that are irrelevant to the property being checked remain abstract during the course of the analysis. Although both tools have been successfully used to check the correctness of some substantial C programs, they focus on analyzing control state transitions of the code (for example, an acquired lock should not be acquired again); they cannot handle the rich data structure properties that we consider.

Saturn [62] also provides a scalable technique for checking temporal safety properties without requiring user-provided intermediate annotations. The scalability is a result of summarizing the called procedures in advance. Given a temporal safety property, Saturn efficiently computes summaries for all called procedures in the form of finite state machines during a pre-process. These summaries are then used in checking the selected procedure against the given property in a one-pass analysis. Saturn has been successfully used to detect memory leaks and erroneous sequences of locks and unlocks in the Linux kernel. However, again, it is highly-tuned for checking finite state properties, and cannot be used for analyzing data structure properties.

On the other hand, other tools are available that focus on checking data structure properties of programs. TVLA [51], for example, has been successfully used to verify insertion sort and bubble sort [40] as well as some properties of lists and binary search trees [48]. It represents the heap as a graph in which the nodes represent objects and edges represent field relations. Since the graph can be arbitrarily large, they abstract it by merging all the nodes equivalent under a particular set of shape predicates into a single abstract node. Although the resulting abstract domain is finite, it may still become very large, and thus the approximating algorithms are too complex to scale to large programs. Furthermore, the precision of the analysis depends on the set of predicates of interest provided by the user.

Hob [37], and Jahob [35] also focus on checking data structure properties. Hob provides a suite of theorem provers to check the verification conditions automatically generated from the code. It has been successfully used to verify a number of applications (e.g. an HTTP server and an implementation of the minesweeper game). Its analysis process, however, depends on user-provided procedure specifications and loop invariants. Hob assumes a simplified object-oriented language with no inheritance and polymorphism, and only handles set-based data structure properties – those that can be expressed by the boolean algebra of sets. Therefore, although it can handle properties about the contents of data structures (e.g. exposed cells are disjoint from mined cells in a minesweeper game), it does not handle, for example, those involving relationships between objects (e.g. whether a key object is linked to a value in a map data structure), or orderings of objects (e.g. in a list data structure). Jahob is a successor of Hob that handles a richer programming language as well as richer specifications. But it still requires user-provided procedure specifications and loop invariants in order to generate verification conditions from programs.

Inferring the behavior of procedures automatically is not a new concept. Interprocedural data flow analysis techniques, in particular, typically summarize the behavior of the called procedures in order to extract different pieces of information about a top level procedure. Points-to analysis [9, 50, 61], for example, computes the set of objects that a pointer can possibly point to in different executions of the code. A procedure summary in this case consists of the pointer information of a procedure. Side-effect analysis [10, 43, 52] is another example of data flow analysis that detects what heap locations may be mutated by each procedure. This information can be stored as a procedure's summary and be reused at the call sites. While such summarization improves the analysis by preventing it from computing the behavior of a called procedure at every call site, it is applied to extract only certain information from the code. That is, such procedure summaries are not sufficient for checking arbitrary data structure properties of the procedures.

1.2 Proposed Solution

Modularization of a program into procedures provides natural abstraction boundaries that make reasoning about the program's behavior much easier. We introduce a fully automatic technique that exploits the modularity of the code in order to provide a modular analysis. That is, a procedure is checked against a given property by substituting specifications for its called procedures.

However, rather than being provided by the user, the specifications are inferred from the code automatically. These specifications are exploited in checking like the specifications of Boogie, but are refined by a mechanism more similar to that of SLAM and BLAST.

1.2.1 Specification Refinement

The fundamental idea underlying our technique is a familiar one: counterexample-guided refinement of an abstraction [11]. The general scheme is as follows:

1. For a program P, generate an abstraction $A(P)$ that overapproximates P.

2. Check if the property holds in $A(P)$. If no counterexample is found, the analysis terminates and has successfully verified the code (against the given property).

3. If a counterexample is found, it is checked for validity. If it is valid, a fault has been discovered and the analysis terminates.

4. Otherwise, the abstraction $A(P)$ is refined so that the invalid counterexample is eliminated. The process then starts over at step 2.

This scheme provides a fully automatic analysis that holds promise for scalability. It has been previously applied in a number of different contexts [4, 19, 20, 26]. Our approach, however, differs from all the previous work in that the abstraction and its subsequent refinements follow the abstraction boundaries of the code itself. To our knowledge, all previous applications of this idea to software analysis involve refinement of predicate abstractions [25]. Our approach, in contrast, refines the specifications used to represent the behavior of called procedures.

We introduce a framework for counterexample-guided refinement of procedure specifications. It starts with a rough initial specification for each called procedure and refines it on demand in response to spurious counterexamples. The framework assumes an underlying analysis in which counterexamples are found by solving a logical formula extracted from the code and the property. The property can be expressed in any language that can be converted to logical constraints. The validity of the found counterexamples are checked again using a constraint solver, and specifications are refined based on the proof of unsatisfiability generated by the solver when the counterexample is invalid. The framework requires the logic to be *decidable*. That is, the validity of the logical formulas can be determined in finite time. Our implementation uses the propositional logic underlying the Alloy modeling language [30] as the logic, and a SAT solver as the constraint solver.

The analysis can start with arbitrarily weak initial specifications. Even empty specifications that allow the procedures to arbitrarily mutate the state of the program can be used in the initial abstraction. However, starting from richer specifications usually reduces the number of refinements needed to check the property, and thus improves the performance.

The main challenge in computing the initial abstraction is to extract specifications that are safe – that is, accounting for all possible executions of the code – and compact – that is, easy to analyze – and yet not too approximate – that is, giving useful information about the underlying code.

We introduce a lightweight technique for extracting an initial specification for a procedure, that computes symbolic bounds on the final values of each field and variable that may be mutated by that procedure. The specifications inferred by this technique do not depend on the property being checked; they give general abstractions of a procedure's behavior that can be used in any context.

This initial specification extraction technique is an application of the abstract interpretation framework [13] in which the abstract domain consists of relational expressions. A procedure is evaluated symbolically, with each field and variable initially holding a value represented by a fresh constant. Each statement updates an environment mapping fields and variables to values represented as relational expressions. After an if-statement, the branches are merged by forming a union expression. A loop is handled by computing a fixpoint which often results in an expression involving a transitive closure. The analysis maintains both upper and lower bounds on the values of variables and fields, so that in some cases an exact result can be obtained.

1.2.2 Instantiation: Karun

We have implemented our specification abstraction refinement technique in a tool called Karun. It checks an object-oriented program written in a subset of Java against a property expressed in Alloy [30].

Alloy is a first order relational logic with a transitive closure operator, making it well-suited for expressing data structure properties succinctly. A constraint on the heap of an object-oriented program can be expressed in Alloy by adopting a *relational view* of the heap: a field `f` of type `T` declared in a class `C` is viewed as a relation `f: C -> T` mapping objects of type `C` to objects of type `T`. Field dereferencing becomes relational join, and reachability becomes a formula involving a transitive closure.

The specifications inferred during the course of the analysis are also expressed in Alloy. The initial abstraction computes symbolic bounds on the final values of each relation (fields and variables) in the form of Alloy relational expressions. Karun translates the given code, the extracted specifications, and the negation of the property being checked to a boolean formula in conjunctive normal form (CNF) using Forge [14]. It then uses the ZChaff SAT Solver [45] to find a solution to the generated formula. The validity of the found solutions is checked using ZChaff again. If a solution represents an invalid counterexample, ZChaff generates a proof of unsatisfiability called an *unsat core* [63] which is an unsatisfiable subset of the solved formula, given as a witness to the unsatisfiability of that formula. The unsat core is then used to generate a refined Alloy specification for a procedure call so that the invalid counterexample is eliminated. It should be noted that Karun is not restricted to use ZChaff; any other SAT solver that provides a proof of unsatisfiability for unsatisfiable formulas can be used as its solving engine.

In order to translate Java programs to finite boolean formulas, Forge finitizes the length of the program executions. That is, it unrolls loops using some user-provided bound. Furthermore, since Alloy is a first order logic, translation of an Alloy constraint to a boolean formula is based on bounding the size of the relations in that Alloy constraint. Therefore, the analysis also requires a bound on the number of objects of each datatype. Consequently, Karun analyzes programs with respect to a bounded scope: the user has to provide a bound on the number of loop iterations, and some bounds on the size of the heap. As a result, although the returned counterexamples are guaranteed not to be spurious, their absence does not constitute proof of correctness.

1.2.3 Limitations

Our analysis framework is not designed for checking full specifications of programs. Given a property, our context-dependent analysis extracts those parts of the called procedures that are relevant to the property; all irrelevant parts remain abstract. Therefore, this analysis is most beneficial when only a subset of the code has to be analyzed. Although the technique can be used to check the full behavior of a program, it may require several specification refinements to infer the full behavior of all procedures, and thus the benefit of the approach may be lost. A simple analysis that checks the whole code without iterations (e.g. by inlining all called procedures) may in fact perform better in such cases.

Although not inherent in the analysis framework, our instantiation of the framework, Karun, has a few limitations. The most important one is that it does not perform a complete analysis. That is, it misses those bugs in the program that require more loop iterations and/or objects than the bounds considered during the analysis. However, the analysis is exhaustive with respect to those bounds. That is, any bugs within the bounds are guaranteed to be found by Karun. Therefore, although Karun does not perform a complete verification, its analysis can be categorized as *bounded verification.*

Furthermore, Karun currently handles only a subset of the Java language, including basic statements, inheritance, and dynamic dispatch; it does not support arrays, exceptions, multithreading, and any numerical expressions other than integer additions and subtractions. One-dimensional arrays, exceptions, and full integer arithmetic have been recently added to the Forge language and will be incorporated into Karun in the future. However, multi-dimensional arrays, multithreading constructs, and non-integer numerical expressions remain topics for future research.

Regarding properties, Karun handles the full Alloy language. Thus, any constraint expressible in Alloy can be analyzed by Karun. However, there are certain properties that cannot be expressed in Alloy, and thus checked by Karun. A functional correctness property that requires higher-order quantification is an example. To illustrate this, consider a method that takes a linked list and makes it "well-formed" (based on some definition of well-formedness) by updating the `next` fields of its entries. We can check that the method is correct by analyzing the property `wellFormed(next')` where `next'` is the resulting `next` link in the post-state. However, if we would like to check that the modified list is the closest well-formed list to the original one (based on some definition of distance), we should analyze the following property:

```
wellFormed(next') and
all next0: Entry -> Entry | wellFormed(next0) =>
        (distance(next, next0) >= distance(next, next'))
```

This property, however, requires support for higher-order quantification because of the quantifier over the relation `next0`. Therefore, it cannot be analyzed by Karun. There are also some categories of constraints, e.g. liveness, time-based, and performance properties, that cannot be analyzed using Alloy; checking them would require a different kind of logic.

1.3 Example

In this section, we illustrate our technique by first describing the user's experience checking the Java code given in Figure 1-1, and then describing the underlying analysis briefly.

The code is an implementation of a topological sort algorithm on a directed graph. The graph is represented by a list of nodes where each node contains two adjacency lists corresponding to its incoming and outgoing edges (stored in the `inNodesList` and `outNodesList` fields, respectively). The sizes of these lists are given by the `inDeg` and `outDeg` fields.

The `topologicalSort` method permutes the nodes of a graph to form an order in which any node n succeeds all the nodes that have an edge going to n. The algorithm is a simple one: it iteratively chooses a source node, i.e. a node with no incoming edges, and removes its outgoing edges from the list of the incoming edges of the remaining nodes. However, to avoid mutating the adjacency lists in the original graph, this example uses an auxiliary field, `visitedInsNum`, that keeps track of the number of nodes from the `inNodesList` that have been already put in the topological sort. A node is declared as a source node when all the nodes in its `inNodesList` are already visited. That is, `inDeg == visitedInsNum`. The `topologicalSort` method returns `true` if the graph is acyclic and the topological sort is computed successfully. If the graph contains a cycle, at some point during the algorithm, no source nodes can be found. The method returns `false` in this case.

We analyze the `topologicalSort` method against the properties given in Figure 1-2. The first property, given in Figure 1-2(a), specifies that if the algorithm terminates

```
class List {
  ListEntry head;
}

class ListEntry {
  Node node;
  ListEntry next;
  ListEntry prev;
}

class Node {
  List inNodesList;
  List outNodesList;
  int inDeg;
  int outDeg;
  int visitedInsNum;
}

class Graph {
  List nodes;

  boolean topologicalSort() {
    boolean isAcyclic = true;
    init();
    ListEntry cur = nodes.head;
    while (cur != null) {
      ListEntry source = findSource(cur);
      if (source == null) {
        isAcyclic = false; break;
      }
      fixIns(source.node);
      Node tmp = source.node;
      source.node = cur.node;
      cur.node = tmp;
      cur = cur.next;
    }
    return isAcyclic;
  }
```

```
void Graph.init() {
  ListEntry c = this.nodes.head;
  while (c != null) {
    c.node.visitedInsNum = 0;
    c = c.next;
  }
}

ListEntry Graph.findSource(ListEntry entry) {
  ListEntry e = entry;
  while ((e != null) &&
   (e.node.inDeg != e.node.visitedInsNum))
    e = e.next;
  return e;
}

void Graph.fixIns(Node n) {
  ListEntry p = n.outNodesList.head;
  while (p != null) {
    p.node.visitedInsNum =
          p.node.visitedInsNum + 1;
    p = p.next;
  }
}
```

Figure 1-1: Example.

successfully (i.e. when the method returns `true`), the computed topological sort is in fact a permutation of the nodes of the original graph. The second property, given in Figure 1-2(b), states that if the algorithm terminates successfully, the in-degree of each node is at most equal to the number of nodes preceding it in the computed topological order. We use primed and unprimed field names to denote the values of the fields after and before the execution of the method, respectively.

To ensure the consistency of the data stored in different data structures, the properties enforce the representation invariants given in Figure 1-2(c). The invariants specify that: (1) the graph does not contain any hyper-edges, i.e. none of the nodes in the graph has more than one outgoing edge to any other node, (2) the `inDeg` field of each node gives the number of nodes in its `inNodesList`, (3) the `outDeg` field gives the number of nodes in the `outNodesList` of each node, (4) if a node `n1` is in the `inNodesList` of a node `n2`, then `n2` should be in the `outNodesList` of `n1`, and (5) the `prev` link must be the inverse of the `next` link. The auxiliary value `contents` represents the set of the nodes stored in a list, obtained by the expression `head.*next.node` which gives the `node` field of all the entries reachable from the head of the list by following the `next` link. (The operator '*' is a transitive closure operator that represents reachability.) Constraints within a pair of curly braces are implicitly conjoined.

In order to perform the analysis, the user has to provide two types of bounds: a bound on the number of loop iterations, and a bound on the heap size. In this example, we analyze the code with respect to 2 loop unrollings, 2 objects of each type, and a maximum bitwidth of 3 for integers.

1.3.1 User Experience

Valid properties: Both the permutation and the in-degree properties hold in the code given in Figure 1-1. Therefore, checking the `topologicalSort` method against these properties does not generate any counterexamples, implying that the property holds within the analyzed bounds.

Invalid properties: If the code were buggy, a counterexample would have been

```
((repInv = true$rel) and (topologicalSort$return = true$rel)) =>
   (this.nodes'.head'.*next'.node' = this.nodes.head.*next.node)
                                                        permutation property
```

(a)

```
((repInv = true$rel) and (topologicalSort$return = true$rel)) =>
   (all e: this.nodes'.head'.*next' | int(e.node'.inDeg) < size(e.*prev'))
                                                          in_degree property
```

(b)

```
repInv =
  let contents = head.*next.node in {
   (all n:this.nodes.contents | all e1, e2:  n.outNodesList.head.*next |
     (e1 != e2) => (e1.node != e2.node))                        no hyper_edge
   (all n:  Node |
     n.inDeg = size(n.inNodesList.contents))                 correct in_degree
   (all n:  Node |
     n.outDeg = size(n.outNodesList.contents))              correct out_degree
   (all n1, n2:  Node |
     (n1 in n2.inNodesList.contents) <=> (n2 in n1.outNodesList.contents))
                                                     consistent adjacency lists
   (all e1, e2:  ListEntry |
     (e1.next = e2) <=> (e2.prev = e1))                        consistent links
  }
```

(c)

Figure 1-2: Properties: (a) the resulting order of nodes is a permutation of the original nodes of the graph, (b) the in-degree of each node is smaller than the number of nodes preceding it in the topological sort, (c) the representation invariants.

generated. In order to illustrate that, we seed a bug in the code of Figure 1-1: we assume that the first statement of the init method is mistakenly written as c = nodes.head.next rather than c = nodes.head. The in-degree property does not hold any more, and the counterexample given in Figure 1-3 is generated. It should be noted that although the bug is in a called method, because it prevents the top level method from satisfying the property, it can be caught by our technique. Our tool currently outputs counterexamples in a textual format similar to the one shown in Figure 1-3(a). It can be further improved to generate a graphical output as shown in Figure 1-3(b) and highlight the program statements executed in the error trace as shown in Figure 1-4.

Figure 1-3(a) shows the values of the program variables and fields in the pre- and post-states. The value G0 is a symbolic graph object representing the receiver graph. The values L0, E0, and N0 are symbolic objects of types List, ListEntry, and Node, respectively. The pre-state encodes the heap given in Figure 1-3(b), and corresponds to the graph of Figure 1-3(c).

The values stored in the post-state are the same as those in the pre-state except for N0.visitedInsNum. The post-state violates the in-degree property because E0.node.inDeg is the integer 1, but the size(E0.*prev) is 0. Therefore, the constraint int(E0.node.inDeg) < size(E0.*prev) evaluates to false. In fact, the topologicalSort method should not even return true in this case because the input graph contains a cycle. However, because of the bug in the init method, it returns true.

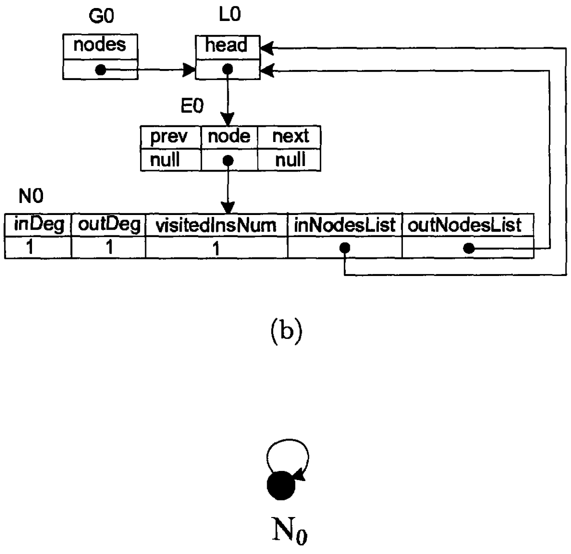
Figure 1-4 gives the program execution corresponding to this counterexample. The statements given in bold represent the executed statements. They are annotated by the values of the variables and fields before and after their execution. To simplify the representation, instead of showing the values of all variables and fields at each point, this figure only gives the updates after each statement. The executions of the called methods are shown by expanding the body of each called method as a block with a smaller font underneath its call site. Actual parameters are substituted for formal parameters. In this example, local variables of called procedures have unique pre-state:

(this = G0) (G0.nodes = L0) (L0.head = E0) (E0.next = null) (E0.prev = null)
(E0.node = N0) (N0.inDeg = 1) (N0.outDeg = 1) (N0.inNodesList = L0)
(N0.outNodesList = L0) (N0.visitedInsNum = 1)

post-state:

(G0.nodes = L0) (L0.head = E0) (E0.next = null) (E0.prev = null)
(E0.node = N0) (N0.inDeg = 1) (N0.outDeg = 1) (N0.inNodesList = L0)
(N0.outNodesList = L0) (N0.visitedInsNum = 2)

(a)



(b)

(c)

Figure 1-3: Counterexample: (a) textual description, (b) pre-state heap, (c) corresponding graph.

```
boolean Graph.topologicalSort() {
[(this = G0), (G0.nodes = L0), (L0.head = E0), (E0.next = null), (E0.prev = null)
(E0.node = N0), (N0.inDeg = 1), (N0.outDeg = 1), (N0.inNodesList = L0),
(N0.outNodesList = L0), (N0.visitedInsNum = 1)]
 boolean isAcyclic = true;                                   [.., isAcyclic = true]
 init();
    ListEntry c = this.nodes.head.next; //bug seeded                  [.., c = null]
    while (c != null) {
      c.node.visitedInsNum = 0;
      c = c.next;
    }
 ListEntry cur = nodes.head;                                         [.., cur = E0]
 while (cur != null) {
   ListEntry source = findSource(cur);
      ListEntry e = cur;                                               [.., e = E0]
      while ((e != null) && (e.node.inDeg != e.node.visitedInsNum))
         e = e.next;                                                 [.., e = null]
      return e;                                                   [.., source = E0]
   if (source == null) {
     isAcyclic = false;
   } else {
     fixIns(source.node);
        ListEntry p = source.node.outNodesList.head;                   [.., p = E0]
        while (p != null) {
          p.node.visitedInsNum = p.node.visitedInsNum + 1;  [.., N0.visitedInsNum = 2]
          p = p.next;                                                [.., p = null]
        }
     Node tmp = source.node;                                         [.., tmp = N0]
     source.node = cur.node;                                     [.., E0.node = N0]
     cur.node = tmp;                                             [.., E0.node = N0]
     cur = cur.next;                                               [.., cur = null]
   }
 }
 return isAcyclic;                                            [.., return = true]
[(G0.nodes = L0), (L0.head = E0), (E0.next = null), (E0.prev = null),
(E0.node = N0), (N0.inDeg = 1), (N0.outDeg = 1), (N0.inNodesList = L0),
(N0.outNodesList = L0), (N0.visitedInsNum = 2)]
}
```

Figure 1-4: Program execution corresponding to the counterexample. Executed statements are given in bold. Called methods are expanded below call sites. Program states before and after the execution, and the state updates after each executed statement are also given.

names, thus used directly. In an actual implementations, however, the names are preceded by unique scope identifiers to avoid name clashes. All loops are iterated at most once.

In the rest of this section, we describe analyzing the code given in Figure 1-1 where both properties hold. Analyzing the buggy case follows the same general steps, but infers different specifications.

1.3.2 Underlying Analysis

Our tool, Karun, analyzes the `topologicalSort` method against each property by inferring specifications for its call sites. Although at the end of the analysis, the inferred specifications are not necessarily complete, they encode sufficient information about the called procedures to check the property. When the analyzed property is partial, our approach, in general, performs substantially better than inlining the method calls. Checking the `topologicalSort` method against the in-degree property (with respect to 3 loop unrollings, 3 objects of each type, and a maximum bitwidth of 4), for example, takes 47 seconds if the call sites are inlined. Karun, however, performs this analysis in 21 seconds. Similarly, checking this method against the permutation property (with respect to 4 loop unrollings, 4 objects of each type, and a maximum bitwidth of 5) by inlining takes 51 seconds whereas Karun's analysis takes only 3 seconds. The rest of this section briefly describes how specifications are inferred, and illustrates the analysis for 2 loop unrollings, 2 objects of each type, and a bitwidth of 3 for integers.

The analysis starts by first computing initial specifications for the methods reachable from the `topologicalSort` method. Figure 1-5 gives the computed specifications. As mentioned before, these specifications do not depend on the property being checked. Chapter 4 explains in detail how they are computed automatically from the code, and what exactly they mean. Here we only give an informal description. The specification computed for the `init` method is its full specification. It denotes that the `init` method changes the `visitedInsNum` field of all the nodes of the graph to zero. The initial specifications of `fixIns` and `findSource`, on the other hand, are

```
void Graph.init():
  visitedInsNum' = visitedInsNum ++ (this.nodes.head.*next.node -> 0)

void Graph.fixIns(Node n):
  visitedInsNum' ⊆ visitedInsNum + (n.outNodesList.head.*next.node -> Int)
  visitedInsNum' ⊇ visitedInsNum - (n.outNodesList.head.*next.node -> Int)

ListEntry Graph.findSource(ListEntry entry):
  findSource$return ⊆ (entry.*next &
                       (null + node.inDeg.(entry.*next.node.visitedInsNum)))
  findSource$return ⊇ none
```

Figure 1-5: Initial specifications.

approximate. The specification for `fixIns` specifies that given an argument node `n`, this method may change the `visitedInsNum` field of any node in the `outNodesList` of `n` to an arbitrary integer, and the specification for `findSource` specifies that this method can return any list entry reachable from the argument list entry `entry`, which is either `null` or its node has an `inDeg` field equal to the `visitedInsNum` field of some node in the entries reachable from `entry`.

Using these initial specifications, the analysis checks `topologicalSort` against the permutation property using a constraint solver. In this case, the solver does not find any counterexamples, meaning that the property holds in the abstract program, and thus, in the original program. Therefore, the analysis terminates and the property has been validated for this finite domain. The fact that analyzing this property does not require any specification refinements implies that the specifications that are initially extracted are accurate enough, i.e. they encode enough information about the called procedures, to validate this property.

However, checking the second property requires some specification refinements. Similar to checking the previous property, checking this property starts using the initial specifications given in Figure 1-5 for the called methods. In this case, the constraint solver finds a counterexample. But the counterexample is not valid in the original program due to an invalid state transition assigned to the first call to the `fixIns` method. Therefore, the specification for `fixIns` is refined. Chapter 6 describes in detail how a counterexample is checked for validity, and how the results

```
Graph.fixIns(n):
 visitedInsNum' =
  if (n.outNodesList.head != null$rel)
  then if (n.outNodesList.head.next != null$rel)
    then visitedInsNum'
    else visitedInsNum ++ (n.outNodesList.head.node ->
                        (int(n.outNodesList.head.node.visitedInsNum) + 1))
  else visitedInsNum'
```

Figure 1-6: Final specification for `fixIns`.

of that check are used to refine the specifications.

As a result of this refinements, the specification given in Figure 1-6 is inferred. It specifies only a partial behavior of the `fixIns` method: if the `outNodesList` of the argument node `n` contains only one node (that is, `(n.outNodesList.head != null)` but `(n.outNodesList.head.next = null)`) then the `visitedInsNum` field of that node is incremented by one. In all other cases, the value of the `visitedInsNum` field is unconstrained.

Although the inferred specification is very partial, it is still sufficient to check the given property. That is, after conjoining this inferred specification with the previous specification of the first call to `fixIns`, no further counterexamples are found. Therefore, the analysis terminates without further specification refinements, and the property has been validated in the analyzed domain.

As shown by this example, when checking partial properties of a program, we do not necessarily need the full specifications of all methods. For example, although the initial specifications of `fixIns` and `findSource` were very weak, they were still sufficient for checking the permutation property. The number of iterations that the analysis goes through totally depends on the property: the permutation property, for example, was validated with only the initial iteration of the analysis whereas the in-degree property required one more iteration to refine some specifications. Furthermore, as shown by this example, the inferred specifications not only depend on the property being checked, but also on the call site: in checking the in-degree property, the first call to `fixIns` required a specification refinement, but for the subsequent calls to this method the weak initial specification sufficed.

1.4 Evaluation

We have evaluated our technique on two case study applications, involving the verification of two open source APIs: a graph package, and a job scheduler.

The graph package, OpenJGraph [1], is a Java implementation of a graph API that provides a number of algorithms (e.g. graph traversals, shortest paths, and minimum spanning trees) for directed and undirected graphs. We have checked 20 methods in two units of this package. Because the API implements well-known algorithms, we were able to extract the specifications from a widely used algorithms textbook [12]. In addition to that, we have checked the code with respect to some representation invariants extracted from the informal comments available in the code. All checks of this package have successfully passed.

The second case study involves checking a job scheduling API called Quartz [2] which is claimed to be used by thousands of people. We have checked 38 methods in 4 units of this package with respect to the correctness properties extracted from the informal comments available in the code.

Our experiments have uncovered two previously unknown bugs in Quartz that are actually observable by its users. Both bugs involve deeply structural properties of the code, and are caused by the errors in some called methods rather than the analyzed top-level method. In particular, one of the errors is in an overriding method which may be called by dynamic dispatch of a call site.

This experiment shows that although the user does not provide any specifications for the called procedures, our technique can still find those errors in the called procedures that are relevant to the property being analyzed. This is because the inferred specifications completely conform to the code of the called methods, and thus will carry all the errors relevant to the analyzed property.

Furthermore, although more experiments are needed to evaluate the scalability of this technique, our current experiments suggest that the technique can handle procedures with a realistic number of calls. In fact, one of the methods checked in the OpenJGraph package invoked 81 other methods directly or indirectly.

We also performed some experiments to evaluate the performance of our analysis technique. We compared our technique with our implementation of another technique that inlines all call sites, rather than abstracting them. Therefore, the results of this comparison show how the on-demand specification inference affects the performance of the analysis. Our experiments suggest that our technique substantially reduces the analysis time when the analyzed property is partial. However, in checking full properties of the code, a simple analysis that inlines all procedures performs better.

We performed other experiments to evaluate our abstract interpretation technique that extracts initial procedure summaries. We evaluated the accuracy of the extracted summaries by checking if the summaries generated for a number of procedures were sufficient to prove their specifications. In 17 of the 30 procedures, the generated summaries were sufficient to check the full specifications; in 16 procedures, the generated summaries included all frame conditions, and provided some partial specifications. Only in one case there was a significant loss of information.

We also evaluated the effectiveness of our abstract interpretation technique by comparing the extracted summaries against *frame conditions* – specifications that only constrain which variables and fields are not modified by a procedure. Our experiments show that in almost all cases the summaries extracted by abstract interpretation result in a substantially reduced analysis time. This implies that not only are they effective in ruling out many invalid executions of the abstract code, but also the overhead of extracting these summaries is negligible in the overall analysis time. These results suggest that our abstract interpretation technique represents a useful balance between tractability and accuracy.

1.5 Contributions

This thesis makes the following contributions:

- **A Context-Sensitive Specification Inference Technique.** This thesis introduces a technique to analyze a procedure against a data structure property by inferring property- and call site-dependent specifications for the called procedures. All previous techniques for checking rich data structure properties of code either require extensive user-provided call site specifications (sometimes even loop invariants), or inline the code of all called procedures. Therefore, their scalability to large pieces of code is limited.

Our technique provides potential scalability by applying the counterexample-guided abstraction refinement framework. To our knowledge, this is the first application of this framework to infer procedure specifications that are suitable for checking rich data structure properties. All previous applications of this framework refine predicate abstractions of the code, which requires a user-provided set of initial predicates.

- **A Stand-Alone Specification Extraction Technique.** This thesis also introduces a technique to extract specifications statically from the text of a program. These specifications are syntactic objects expressed in the Alloy language, readable by both tools and users. Although designed to be used as the initial abstraction of the code in our abstraction refinement framework, the specifications neither are context-dependent, nor require any code finitizations. They summarize the behavior of a procedure in general. Therefore, they can be used as stand-alone approximations of the code in a variety of settings.

  The algorithm to generate these specifications is scalable: it is linear in the number of fields and the size of the code. The technique is an application of the abstract interpretation framework where the abstract values are widened based on some user-provided parameters. Therefore, the precision of the generated specifications can be tuned by the users.

- **An Analysis Framework.** This thesis describes the specification inference idea in the form of an abstract analysis framework. The framework assumes an underlying analysis in which counterexamples are found by solving constraints extracted from the code and the given property. The following features of the framework makes it easier to exploit the idea in a wider range of settings:

   1. It assumes very little about the programming language except that it supports procedure declaration.

   2. It does not assume any particular translation of code to constraints, except that the generated constraints are finite and can be solved by some constraint solver.

   3. It provides rigorous conditions for termination, soundness, and completeness.

  Although Karun, our proposed instantiation of the framework, requires finitization of the program executions and the size of the heap, the framework itself does not depend on these compromises, and seems to hold promise for application in other contexts, such as the method proposed by Flanagan [19].

- **A Bug Finding Tool.** This thesis introduces an instantiation of the framework, Karun, as a SAT-based program analysis prototype tool that checks rich data structure properties of Java programs with respect to a finite domain. The tool substantially reduces the human cost involved in formal program analysis by providing the following two features:

   1. It does not require any user-provided annotations beyond the top-level property being checked.

   2. It never generates spurious counterexamples. That is, any counterexample returned to the user is a real execution of the code that violates the property being checked. Therefore, it relieves the user from going over a list of warnings for possible bugs.

- **Justification: Case Studies.** We have evaluated our technique on two case study applications: a graph package, OpenJGraph, and a job scheduler, Quartz. We have checked the procedures of these packages against some data structure properties that specify correctness conditions of the code. Our experiments have uncovered two previously unknown bugs in Quartz that are actually observable by its users. Both bugs are caused by the errors in called methods rather than the analyzed top-level method.

Our case studies illustrate that our technique can handle realistic Java code that makes many calls and accesses several different data structures. Although users are not required to provide specifications for the call sites, our analysis is capable of finding errors not only in the top-level procedure being analyzed, but also in the called procedures if the errors prevent the top-level procedure from satisfying the analyzed property. Our experiments suggest that our specification refinement technique substantially reduces the analysis time (compared to a technique that inlines call sites) when the analyzed property is partial. However, in checking full properties of the code, a simple analysis that inlines all procedures is more effective.

1.6 Thesis Organization

This thesis is organized as follows: Chapter 2 provides a formal description of our specification inference technique and proves its main properties. This chapter describes the analysis as a general framework parameterized over different functions, allowing it to be instantiated in a variety of settings.

Chapters 3 to 6 describe our instantiation of the framework as implemented in our analysis tool, Karun. Chapter 3 gives the analysis context: the supported programming language, the specification language, and the required analysis parameters.

Chapter 4 introduces our technique for summarizing the behavior of procedures. This technique produces procedure specifications that will be used in the initial abstraction of the code.

Chapter 5 presents Forge, the engine used by Karun to analyze a given procedure against a given property. It describes how Karun uses Forge to generate an abstract program from the original code and the specifications extracted for its call sites, and how the abstract program is checked for a counterexample.

Chapter 6 gives the details of checking the validity of a found counterexample, and using the results to refine the specification of some procedure in the abstract program.

Chapter 7 gives the experimental results, including two case studies, and different experiments performed to evaluate the technique.

Chapter 8 concludes the thesis by discussing different aspects of our technique, comparing it with the related work, and highlighting some possible directions for future research.

Chapter 2

Method

This chapter explains the essence of our technique formally and discusses its termination, soundness, and completeness properties. The technique is described as an abstract framework for checking a procedure against a user-provided property. The property can be expressed in any language as long as it can be translated to a logical formula. The framework is parameterized by basic functions, assumed to satisfy some basic axioms, in order to make the main idea as general as possible. Chapters 3 to 6 will provide concrete examples for the basic functions introduced here.

2.1 Overview

Our framework provides a scalable modular analysis in which the behavior of the called procedures are automatically inferred from the code as needed to check a given property. The effects of each call site on the program state are initially abstracted by some rough specification. This specification is later refined if shown to be so approximate that the analysis finds invalid counterexamples due to the abstract behavior at that call site. The analysis terminates when either a valid counterexample is found or it is shown that no counterexamples exist. Figure 2-1 shows the analysis framework. It consists of the following phases:

Abstraction: The body of the procedure selected for analysis is translated to a logical formula. This formula captures the semantics of the procedure exactly except

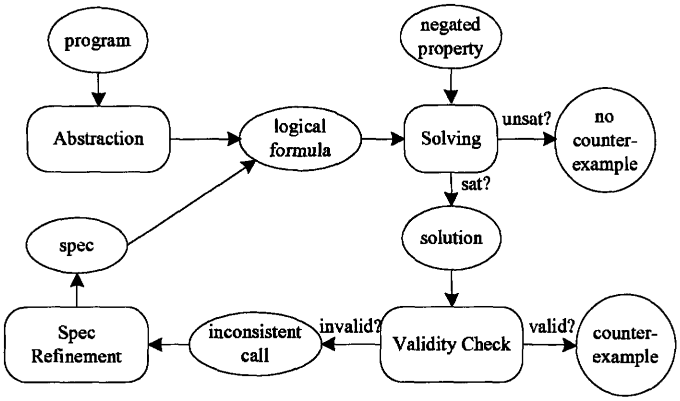


Figure 2-1: An overview of the framework.

at its call sites. All procedure calls are replaced with approximate specifications so that the abstraction is an over-approximation of the original code.

Solving: The generated formula and the negation of the property being checked are handed to a solver. If the formula has no solution, the property holds in the abstract procedure, and since the abstraction is an over-approximation of the original procedure, the property holds in the original procedure too. On the other hand, if a solution is found, it indicates a potential violation of the property, and must be checked for validity.

Validity check: The validity of a solution is determined by checking the consistency of the found solution with each procedure call, again using a solver. If the solution is consistent with all procedure calls, it represents a feasible counterexample, and the analysis terminates. Otherwise, the specification of the inconsistent call must be refined.

Specification refinement: A new partial specification is inferred for an inconsistent procedure call from a proof of invalidity generated by the solver. This specification rules out the given invalid solution. It is conjoined with the old specification of the procedure to form a refined specification. The process then starts over at the solving phase.

Termination, soundness (that is, found counterexamples are feasible), and completeness (that is, all counterexamples are found) of this analysis depend on the

$$\begin{array}{lll}
\texttt{program} & \texttt{::=} & \overrightarrow{\texttt{procedure}} \\
\texttt{procedure} & \texttt{::=} & \texttt{name(}\overrightarrow{\texttt{variable:type}}\texttt{)[:type]}\ \{\overrightarrow{\texttt{statement}}\} \\
\texttt{statement} & \texttt{::=} & \texttt{compoundStmt}\ \mid\ \texttt{elemStmt} \\
\texttt{compoundStmt} & \texttt{::=} & \texttt{if predicate(}\overrightarrow{\texttt{expr}}\texttt{)}\ \overrightarrow{\texttt{statement}}\ \texttt{[else}\ \overrightarrow{\texttt{statement}}\texttt{];} \\
 & \mid & \texttt{while predicate(}\overrightarrow{\texttt{expr}}\texttt{)}\ \overrightarrow{\texttt{statement}}\texttt{;} \\
 & \mid & \texttt{name(}\overrightarrow{\texttt{variable}}\texttt{);}
\end{array}$$

Figure 2-2: Abstract syntax for the analyzed language.

particular translation technique that encodes program statements as logical formulas, and the solver used in the instantiation of the framework. The framework, in general, does not guarantee termination although our instantiation of the framework described in Chapters 3 to 6 is guaranteed to terminate. As proved in Section 2.7, the specification inference approach does not introduce additional unsoundness beyond possible compromises made by the translation technique. Furthermore, if the instantiation guarantees termination of the analysis, the specification inference approach does not introduce additional incompleteness either.

2.2 Definitions

Program syntax. We target imperative programs supporting procedure declarations. Figure 2-2 gives an abstract syntax for a fragment of a basic programming language. A program is a sequence of procedure declarations. A procedure declaration consists of a name, a list of formal parameters, the type of the return value, and a body. The body of a procedure is a sequence of statements that can be either compound or elementary statements. Compound statements can be conditionals, loops, and procedure calls. Elementary statements are updates of variables (locals, procedure arguments, array elements, and fields).

> ***Example.*** We illustrate the concepts described in this chapter using the `xorAll` procedure given in Figure 2-3. This procedure takes a boolean variable `b`, an array of boolean values `a`, and an integer `size` indicating the size of the array, and computes the exclusive or of every element of `a` with `b`.

```
void xorAll(boolean b, boolean[] a, int size) {
  int i = 0;
  while (i < size) {
    a[i] = xor(b, a[i]);
    i = i + 1;
  }
}

boolean xor(boolean x, boolean y) {
  if (x == y) return false;
  else return true;
}
```

Figure 2-3: Example: an `xor` operation.

Program semantics. Let *PVar* be the set of all variables of a program, and *PVal* be the set of all concrete values (e.g. booleans, integers, object addresses) that can be assigned to those variables. A *program state*, $s \in State$, is a partial function from variables to their values:

$$State = PVar \rightharpoonup PVal$$

Each program statement *stmt* is viewed as a relation over *State*:

$$[\![stmt]\!] \subseteq State \times State$$

where $(s, s') \in [\![stmt]\!]$ iff executing *stmt* in the state s can result in the state s'. This definition can be lifted to a sequence of statements inductively as follows:

$$[\![stmt_0; \overrightarrow{stmt}]\!] = \{(s, s') \mid \exists s'', (s, s'') \in [\![stmt_0]\!] \wedge (s'', s') \in [\![\overrightarrow{stmt}]\!]\}$$

A *program trace* is an execution of a program, given as a sequence of pairs of program states and elementary program statements. A special statement, `exit`, is used to represent the of the execution.

$$Trace : \overrightarrow{State \times (ElemStmt \cup \{\texttt{exit}\})}$$

A pair $(s, stmt)$ in a trace t means that the statement *stmt* is executed, and the

```
s = [b -> true, a[0] -> true, size -> 1]
```

(a)

pre-state of the statement:	executed statement:
`s = [b -> true, a[0] -> true, size -> 1]`	**int** i = 0;
`s = [..., i -> 0]`	xor$return = **false**;
`s = [..., xor$return -> false]`	a[i] = xor$return;
`s = [..., a[0] -> false]`	i = i + 1;
`s = [..., i -> 1]`	exit;

(b)

Figure 2-4: Example: (a) an initial program state, (b) the corresponding trace: each line gives an executed statement and the state before the execution of that statement.

program state before the execution of *stmt* is s. Consecutive statements in the trace are executed immediately after each other. It should be noted that the program state of the first pair denotes the initial state of the program, and the program state of the last pair represents the final state of the program. The program statement of the last pair is always `exit`, a dummy statement indicating successful completion of the program.

For a trace t and a program state $s \in t$, let $succ_t(s)$ be the successor of s in t, i.e. the program state immediately following s in t. (The final program state does not have a successor.) A trace t is *valid* if and only if for any pair $(s, stmt)$ included in t, the state transition from s to $succ_t(s)$ is consistent with the semantics of the statement *stmt*. That is,

$$t \texttt{ is valid} \iff \forall (s, stmt) \in t,\ (s, succ_t(s)) \in [\![stmt]\!]$$

Example. Figure 2-4(a) shows an initial program state for the `xorAll` procedure. It maps the variables initially accessible by this procedure to some concrete values. Figure 2-4(b) shows a valid trace of `xorAll` starting from this state. It represents a program execution by enumerating the executed statements and providing their pre-states (the

> post-state of a statement is the pre-state of its subsequent statement). Instead of giving the complete values stored in the program states, this figure only shows the state updates. The executions of the loop and the method call are represented by the elementary statements executed in their bodies. In this example, the loop is executed only once and the call to the `xor` method returns `false`. The return statement is treated as an assignment to the variable `xor$return`.

Variable and value mappings. Our framework relies on translating programs to logical formulas. Any such translation requires a mapping from program variables and their concrete values to logical variables and logical values. We use $\mathit{varMap} \in \mathit{VariableMap}$ and $\mathit{valMap} \in \mathit{ValueMap}$ to represent a variable mapping[1] and a value mapping, respectively:

$$\mathit{VariableMap} = \mathit{PVar} \rightharpoonup \mathit{LVar}$$
$$\mathit{ValueMap} = \mathit{PVal} \rightharpoonup \mathit{LVal}$$

where *LVar* and *LVal* respectively represent the set of logical variables and their possible values.

In order for the translation to be correct, the value and variable mappings must be *valid*. A value mapping is valid iff it is an injective function. That is, it maps each program value to at most one logical value, and maps different program values of the same type to different logical values. A valid variable mapping, however, does not have to be injective. A variable mapping is valid if it is a function (i.e. it maps each program variable to at most one logical variable) that maps different program variables to the same logical variable only if they are guaranteed to be equal in all executions of the program.

> ***Example.*** To illustrate variable and value mappings, we assume a translation function that translates the `xorAll` code to a boolean formula. That is, the logical domain consists of boolean variables and

[1] In general, a variable mapping may map program variables to *expressions* over logical variables. However, for simplicity, we do not discuss that case.

the binary truth values 0 and 1. The following gives as example of a variable mapping and a value mapping:

```
varMap = [b -> v1, a[0] -> v2, a[1] -> v3, a[2] -> v4,
          a[3] -> v5, size -> <v6, v7>, i -> <v8, v9>]
valMap = [true -> 1, false -> 0, 0 -> <0, 0>, 1 -> <0, 1>,
          2 -> <1, 0>, 3 -> <1, 1>]
```

where `v1` to `v9` represent boolean variables in the logical domain. The variable mapping `varMap` encodes each boolean variable in the program domain by a boolean variable in the logical domain. The integer variables `size` and `i` are encoded by pairs of boolean variables in the logical domain, representing a bitwidth of two. Since the array size is represented by two bits, the array can contain at most 4 elements. The value mapping `valMap` maps the concrete values used in the program to some boolean values in the logical domain. The boolean values `true` and `false` in the program domain are encoded by `1` and `0` in the logical domain, respectively. The integer values `0` to `3` are encoded by pairs of boolean values in the binary format.

Logical instances. Our framework assumes that a solver is available to determine the satisfiability of a logical formula and to generate a proof in case of unsatisfiability.

A *logical instance*, $ins \in Instance$, is a function from logical variables to values:

$$Instance = LVar \rightarrow LVal$$

An instance can be *partial*. That is, it can assign values only to some logical variables.

The meaning of a formula $f \in Formula$ is given by the set of logical instances that satisfy f. That is,

$$\llbracket f \rrbracket \subseteq Instance$$

where $ins \in \llbracket f \rrbracket$ if and only if ins is a solution to f, i.e. f evaluates to *true* under the

$$\begin{tikzcd}
\textit{Logic:} & \mathsf{LVar} \arrow[r, "\text{Instance}"] & \mathsf{LVal} \\
\textit{Program:} & \mathsf{PVar} \arrow[u, "\text{Variable Map}"] \arrow[r, "\text{State}"] & \mathsf{PVal} \arrow[u, "\text{Value Map}"']
\end{tikzcd}$$

Figure 2-5: Relationship between different domains.

mapping defined by *ins*. Figure 2-5 shows the relationship between different domains defined in this section.

> ***Example.*** Assume that `v1` to `v9` are boolean variables in a logical domain where the logical values are denoted by `0` and `1`. The following gives a partial logical instance:
>
> ins = `[v5 -> 0, v6 -> 1, v7 -> 0, v8 -> 1, v9 -> 1]`
>
> It maps the boolean variables `v5` to `v9` to some logical values. This instance is partial because it does not constrain the values of the variables `v1` to `v4`.

2.3 Input Functions

Translation. The translation of program statements to logical formulas is represented by the *translate* function. Our framework assumes the existence of such a function and requires it to have the properties discussed below, but is independent of its value.

Given a statement *stmt*, a variable mapping *varMap*, and a value mapping *valMap*, the *translate* function returns a logical formula f that encodes the behavior of that statement. Since the translation of a statement can introduce new logical variables for some program variables, the *translate* function also returns a new variable mapping *varMap*$'$. That is, *varMap* and *varMap*$'$ give the mappings of the variables before and after translating *stmt*, respectively. However, the translation does not introduce new value mappings because a value mapping involves only constant values.

$$\textit{translate} : \textit{Stmt} \times \textit{VariableMap} \times \textit{ValueMap} \rightarrow \textit{Formula} \times \textit{VariableMap}$$

Our analysis framework is independent of how exactly the code is translated. However, it requires the *translate* function to return a *finite* formula for any given piece of code. Furthermore, we assume that *translate* is compositional. That is, it can be lifted to a sequence of statements inductively as follows:

$$\begin{aligned}
&\textit{translate}(\textit{stmt}_0; \overrightarrow{\textit{stmt}}, \textit{varMap}, \textit{valMap}) = \\
&\textbf{let } \textit{translate}(\textit{stmt}_0, \textit{varMap}, \textit{valMap}) = (f_1, \textit{varMap}_1), \\
&\quad \textit{translate}(\overrightarrow{\textit{stmt}}, \textit{varMap}_1, \textit{valMap}) = (f_2, \textit{varMap}_2) \\
&\textbf{in } (f_1 \wedge f_2, \textit{varMap}_2)
\end{aligned}$$

That is, a sequence of statements is translated by first translating the first statement of that sequence and then using the resulting variable mapping in the translation of the rest of the statements.

> ***Example.*** Figure 2-6 gives an example of translating the `xorAll` method to a boolean formula. The code is translated with respect to the initial variable and value mappings shown in Figure 2-6(a). This translation handles loops by unrolling them once and the method calls by inlining. Figure 2-6(b) gives the `xorAll` method after the loop is unrolled and the call to the `xor` method is inlined. Because the loop is unrolled once, only arrays with at most one element are considered.
>
> The translation of each statement generates a new variable mapping and a boolean formula as shown in Figures 2-6(c) and 2-6(d), respectively. To improve readability, instead of giving the complete variable mappings, Figure 2-6(c) only shows how the post-state mapping of each statement differs from the pre-state mapping of that statement. The generated boolean formulas encode the effects of each statement. The translation of the assignment in the first line (`i = 0`), for example,

```
varMap = [b -> v1, a[0] -> v2, size -> <v3, v4>]
valMap = [false -> 0, true -> 1, 0 -> <0, 0>, 1 -> <0, 1>]
```

(a)

```
void xorAll(boolean b,
    boolean[] a, int size) {
  int i = 0;
  if (i < size) {
    if (b == a[i])
      a[i] = false;
    else
      a[i] = true;
    i = i + 1;

    assume (i >= size);

  }
}
```

(b)

```
mapping after translating a stmt:

varMap=[..., i -> <v5, v6>]


varMap=[..., a[0] -> v7]


varMap=[..., a[0] -> v7]
varMap=[..., i -> <v8, v9>]

varMap=[...]
```

(c)

```
(¬v5 ∧ ¬v6) ∧
((¬v5 ∧ v3) ∨ (v5 ∧ v3 ∧ ¬v6 ∧ v4) ∨ (¬v5 ∧ ¬v3 ∧ ¬v6 ∧ v4)) ⟹ (
  (((¬v1 ∧ ¬v2) ∨ (v1 ∧ v2)) ⟹
    (¬v7)) ∧
  (((¬v1 ∧ v2) ∨ (v1 ∧ ¬v2)) ⟹
    (v7)) ∧
  ((¬v8 ∧ v9 ∧ ¬v5 ∧ ¬  v6) ∨ (v8 ∧ ¬v9 ∧ ¬v5 ∧ v6) ∨ (v8 ∧ v9 ∧ v5 ∧ ¬v6)∨
   (¬v8 ∧ ¬v9 ∧ v5 ∧ v6)) ∧
  ((v8 ∧ ¬v3) ∨ (v8 ∧ v3 ∧ v9) ∨ (v8 ∧ v3 ∧ ¬v9 ∧ ¬v4) ∨ (¬v8 ∧ ¬v3 ∧ v9)∨
   (¬v8 ∧ ¬v3 ∧ ¬v9 ∧ ¬v4))
)
```

(d)

Figure 2-6: Translation example: (a) initial variable and value mappings, (b) the transformed `xorAll` method, (c) Variable mappings after translating each statement, and (d) the translation of statements to a boolean formula.

allocates fresh boolean variables `v5` and `v6` for the variable `i` and constrains their values to be 0 (i.e. $(\neg\mathtt{v5} \wedge \neg\mathtt{v6})$). The translation of the branching condition in the second line (`if (i < size)`) constrains `i`, encoded by `<v5, v6>`, to be smaller than `size`, encoded by `<v3, v4>`. This constraint is then used as the antecedent of the formulas encoding the body of the conditional. Other statements are translated similarly. The formulas encoding individual statements are conjoined.

Solving. A solver is used to determine whether or not a logical formula has a solution. The solutions generated by the solver are *total functions*, they assign some logical value to every variable that appears in the analyzed formula.

We assume that if a partial logical instance already exists, a solver can determine if that instance can be extended to a solution to a given formula. For a formula f and a partial instance ins_0, $solve(f, ins_0)$ represents the set of all solutions[2] to f that preserve the mapping already defined by ins_0:

$$solve : Formula \times Instance \rightarrow 2^{Instance}$$

$$solve(f, ins_0) = \{ins \mid ins_0 \subseteq ins \ \wedge \ ins \in [\![f]\!]\}$$

where $2^{Instance}$ denotes the powerset of *Instance*. Thus, $solve(f, ins_0) = \emptyset$ implies that no such solution exists. We use $solve(f)$ for $solve(f, \emptyset)$.

Example. Consider the following boolean formula f that encodes the inner conditional of the `xorAll` method as given in Figure 2-6.

$$(((\neg\mathtt{v1}\wedge\neg\mathtt{v2})\vee(\mathtt{v1}\wedge\mathtt{v2})) \implies \neg\mathtt{v7}) \ \wedge \ (((\neg\mathtt{v1}\wedge\mathtt{v2})\vee(\mathtt{v1}\wedge\neg\mathtt{v2})) \implies \mathtt{v7})$$

Solving f with respect to a partial instance $ins_0 =$ `[v1 -> 1, v2 -> 1]` (corresponding to the case where `b` and `a[0]` are both `true`), results in

[2] It is not necessary for the solver to return *all* solutions to a formula; one solution is sufficient for the analysis. However, defining *solve* to return all solutions simplifies the formalization.

only one solution:

$$\mathit{solve}(f, \mathit{ins}_0) = \{\texttt{[v1 -> 1, v2 -> 1, v7 -> 0]}\}$$

This means that the only solution to f that preserves the given values of `v1` and `v2` will map `v7` to 0 (corresponding to updating `a[0]` to `false`).

Proof generation. We assume that the solver used in our analysis can generate a proof of unsatisfiability when the given formula has no solutions. An *unsatisfiability proof* for an unsatisfiable formula f is a logical consequence of f that is also unsatisfiable and thus is a witness that f does not have a solution. Although f is an unsatisfiability proof of itself, a good proof consists of a weaker formula. We use the function *prove* to represent the unsatisfiability proof returned by the solver[3].

If a formula f is solved with respect to a partial instance *ins* and no solution is found, the unsatisfiability proof, *prove*(f, *ins*), will denote a logical consequence of f which is unsatisfiable with respect to *ins*. That is,

$$\mathit{prove} : \mathit{Formula} \times \mathit{Instance} \longrightarrow \mathit{Formula}$$
$$(\mathit{prove}(f, \mathit{ins}) = f') \;\Rightarrow\; ((f \Rightarrow f') \wedge \mathit{solve}(f', \mathit{ins}) = \emptyset)$$

Example. Consider the formula f from previous example:

$$\texttt{(((}\neg\texttt{v1}\wedge\neg\texttt{v2)}\vee\texttt{(v1}\wedge\texttt{v2))} \implies \neg\texttt{v7)} \wedge \texttt{(((}\neg\texttt{v1}\wedge\texttt{v2)}\vee\texttt{(v1}\wedge\neg\texttt{v2))} \implies \texttt{v7)}$$

If f is solved with respect to the following instance

$$\mathit{ins} = \texttt{[v1 -> 1, v2 -> 1, v7 -> 1]}$$

no solution is found. That is, f is unsatisfiable with respect to *ins*. A

[3] The *prove* function can be *non-deterministic*. That is, different applications of this function to the same arguments may generate different outputs.

small unsatisfiability proof is

$$((\texttt{v1} \wedge \texttt{v2}) \implies \neg \texttt{v7})$$

which is a subset of f that cannot be satisfied by the given instance *ins*. Therefore, f does not have a solution that preserves *ins*.

Initial call site specification. In order to compute the initial abstraction of a procedure, our framework requires some initial specification for each call site reachable by that procedure. We use *computeSpec* to represent the function that computes the initial specification of a procedure call. Because computing an initial specification might introduce a new mapping of program variables to logical variables, *computeSpec* also returns the variable mapping of the post-state.

$$\textit{computeSpec} : \textit{CallStmt} \times \textit{VariableMap} \times \textit{ValueMap} \longrightarrow \textit{Formula} \times \textit{VariableMap}$$

Our framework is independent of the exact value of the *computeSpec* function. It only assumes that *computeSpec* always generates a finite logical formula, and the generated formula *over-approximates* the effects of its corresponding call site. That is, it accounts for all the executions of the called procedure. This property is specified formally in Section 2.5. Any specification that satisfies these conditions can be used as the initial specification in our framework. Therefore, even an empty specification that allows arbitrary change in the state of the program can serve as the initial specification. More precise specifications are inferred during the analysis if needed.

Example. Figure 2-7 gives two examples of initial specifications. Both examples approximate the behavior of the `xor` method using the variable and value mappings given in Figure 2-7(a). They both generate a variable mapping in which the return value of the `xor` method is encoded by a fresh boolean variable, namely `v3`. The first specification (Figure 2-7(b)) is a boolean formula that constrains the return value of `xor` to be 0 if the two given arguments are equal. Otherwise, it is

```
boolean xor(boolean x, boolean y) {
  if (x == y) return false;          varMap = [x -> v1, y -> v2]
  else return true;                  valMap = [false -> 0, true -> 1]
}
```

(a)

first specification:

```
  ((¬v1 ∧ ¬v2) ∨ (v1 ∧ v2)) ⟹ ¬v3     varMap=[x-> v1, y -> v2, xor$return -> v3]
```

(b)

second specification:

```
  true                                varMap=[x-> v1, y-> v2, xor$return -> v3]
```

(c)

Figure 2-7: Examples of initial specification: (a) the xor method and its pre-state mappings, (b) first specification and the post-state variable mapping, (c) second specification and the post-state variable mapping.

unconstrained. The second specification (Figure 2-7(c)) does not constrain the return value of `xor` at all; it is just the boolean formula `true`, allowing the method to return any arbitrary boolean value. Since both of these specifications overapproximate the behavior of the `xor` method, they can be used as its initial specification in our framework.

2.4 Computed Functions

Given the input functions *translate*, *solve*, *prove*, and *computeSpec*, we compute the following auxiliary functions to use in our specification refinement algorithm.

Converting states to instances. A program state s can be encoded as a logical instance using a variable mapping *varMap* and a value mapping *valMap*. We use *toInstance* to represent this encoding.

$$\mathit{toInstance} : \mathit{State} \times \mathit{VariableMap} \times \mathit{ValueMap} \longrightarrow \mathit{Instance}$$

$$\mathit{toInstance}(s, \mathit{varMap}, \mathit{valMap}) = \mathit{valMap} \circ s \circ \mathit{varMap}^{-1}$$

```
varMap = [b -> v1, a[0] -> v2, size -> <v3, v4>]
valMap = [false -> 0, true -> 1, 0 -> <0, 0>, 1 -> <0, 1>]
s = [b -> true, a[0] -> true, size -> 1]
ins = [v1 -> 1, v2 -> 1, v3 -> 0, v4 -> 1]

toInstance(s, varMap, valMap) = ins
toState(ins, varMap, valMap) = s
```

Figure 2-8: An example of converting states to instances and vice versa.

That is, *toInstance* generates a logical instance that maps a logical variable v to a logical value which is computed by first finding the program variable that is mapped to v, then looking up the value of that program variable in the given state, and finally converting the resulting value to a logical value using the value mapping.

It should be noted that if the given variable and value mappings are valid, the *toInstance* operation is a function, i.e. it maps each logical variable to at most one logical value. This is because the definition of validity for value and variable mappings ensures that if the given *valMap* is valid, it is a function, and if the given *varMap* is valid, $s \circ \mathit{varMap}^{-1}$ is a function.

> ***Example.*** Figure 2-8 provides an example of converting states to instances. The state shown in this figure defines a pre-state for the `xorAll` method. Using the given value and variable mappings, this program state is converted to the given instance which maps each logical variable v to $\mathit{valMap}(s(\mathit{varMap}^{-1}(v)))$. The variable `v1`, for example, is mapped to the value 1 because $\mathit{varMap}^{-1}(\texttt{v1}) = \texttt{b}$, $s(\texttt{b}) = \texttt{true}$, and $\mathit{valMap}(\texttt{true}) = 1$.

Converting instances to executions. If a formula f encodes the behavior of a program, any solution to f can be interpreted as an execution of the encoded program. Using a variable and a value mapping, the *toState* function extracts a program state from a logical instance:

$$\mathit{toState} : \mathit{Instance} \times \mathit{VariableMap} \times \mathit{ValueMap} \longrightarrow \mathit{State}$$
$$\mathit{toState}(\mathit{ins}, \mathit{varMap}, \mathit{valMap}) = \mathit{valMap}^{-1} \circ \mathit{ins} \circ \mathit{varMap}$$

That is, the generated program state maps each variable v to a value that is computed by first finding the logical variable corresponding to v in *varMap*, then looking up the logical value of that logical variable in the given instance *ins*, and finally mapping that logical value back to a program value using $valMap^{-1}$.

It should be noted that if the given variable and value mappings are valid, *toState* will be a function.This is because by definition, any logical instance *ins* is a function, and by definitions of validity for variable and value mappings, if *varMap* is valid, it is a function, and if *valMap* is valid, it is injective and thus $valMap^{-1}$ is a function.

Example. The example given in Figure 2-8 also illustrates converting instances to states. It represents converting an arbitrary instance to a pre-state of the `xorAll` method. The state maps each program variable x to $valMap^{-1}(ins(varMap(x)))$. The variable `b`, for example, is mapped to the value `true` because $varMap(\texttt{b}) = \texttt{v1}$, $ins(\texttt{v1}) = \texttt{1}$, and $valMap^{-1}(\texttt{1}) = \texttt{true}$.

Using the *toState* function, we define *toTrace* to compute a trace of a given piece of code using a given logical instance. Generating a trace involves finding the executed statements by evaluating branch conditions, and computing the pre-states of the executed statements (post-states are also computed in this process: the post-state of a statement is the pre-state of the next statement). The definition of *toTrace* is given below:

$$
\begin{aligned}
&\mathit{toTrace} : \mathit{Instance} \times \mathit{Stmt} \times \mathit{VariableMap} \times \mathit{ValueMap} \rightarrow \overrightarrow{\mathit{State} \times (\mathit{ElemStmt} \cup \{\texttt{exit}\})}\\
&\mathit{toTrace}(\mathit{ins}, \mathit{stmt}_0 : \texttt{ElemStmt}, \mathit{varMap}, \mathit{valMap}) = (\mathit{toState}(\mathit{ins}, \mathit{varMap}, \mathit{valMap}), \mathit{stmt}_0)\\
&\mathit{toTrace}(\mathit{ins}, \mathit{proc}(a_1, \ldots, a_n), \mathit{varMap}, \mathit{valMap}) =\\
&\quad \textbf{let } \mathit{varMap}_1 = \mathit{varMap}[\mathit{formal}_1 \mapsto \mathit{varMap}(a_1), \ldots, \mathit{formal}_n \mapsto \mathit{varMap}(a_n)]\\
&\quad \textbf{in } \mathit{toTrace}(\mathit{ins}, \mathit{proc\_body}, \mathit{varMap}_1, \mathit{valMap})\\
&\mathit{toTrace}(\mathit{ins}, \texttt{if } (\mathit{cond})\ \mathit{stmt}_1 \texttt{ else } \mathit{stmt}_2, \mathit{varMap}, \mathit{valMap}) =\\
&\quad \textbf{let } s = \mathit{toState}(\mathit{ins}, \mathit{varMap}, \mathit{valMap}) \textbf{ in}\\
&\quad \textbf{if } \mathit{evaluate}(\mathit{cond}, s) = \mathit{true} \textbf{ then } \mathit{toTrace}(\mathit{ins}, \mathit{stmt}_1, \mathit{varMap}, \mathit{valMap})\\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{else } \mathit{toTrace}(\mathit{ins}, \mathit{stmt}_2, \mathit{varMap}, \mathit{valMap})\\
&\mathit{toTrace}(\mathit{ins}, \mathit{stmt}_0 = \texttt{while } (\mathit{cond})\ \mathit{stmt}, \mathit{varMap}, \mathit{valMap}) =\\
&\quad \textbf{let } s = \mathit{toState}(\mathit{ins}, \mathit{varMap}, \mathit{valMap}) \textbf{ in}\\
&\quad \textbf{if } \mathit{evaluate}(\mathit{cond}, s) = \mathit{true} \textbf{ then } \mathit{toTrace}(\mathit{ins}, \mathit{stmt}; \mathit{stmt}_0, \mathit{varMap}, \mathit{valMap})\\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{else } ()
\end{aligned}
$$

That is, generating the trace for an elementary statement only involves computing the program state before the execution of that statement. The trace of a procedure call is computed by first producing a new variable mapping in which the formal parameters of the called procedure are mapped to the same logical variables as the actual arguments, and then generating a trace for the body of the called procedure. The trace for conditionals is computed by first computing the program state before the execution of the conditional, then evaluating the branching condition in that program state, and finally generating the trace for the appropriate branch. Generating a trace for a loop is similar except that the loop body is iterated until the loop condition evaluates to false.

The trace of a sequence of statements is computed as follows:

$$
\begin{aligned}
&\mathit{toTrace}(\mathit{ins}, \mathit{stmt}_0; \overrightarrow{\mathit{stmt}}, \mathit{varMap}, \mathit{valMap}) =\\
&\quad \textbf{let } \mathit{translate}(\mathit{stmt}_0, \mathit{varMap}, \mathit{valMap}) = (f, \mathit{varMap}') \textbf{ in}\\
&\quad \mathit{toTrace}(\mathit{ins}, \mathit{stmt}_0, \mathit{varMap}, \mathit{valMap}); \mathit{toTrace}(\mathit{ins}, \overrightarrow{\mathit{stmt}}, \mathit{varMap}', \mathit{valMap})
\end{aligned}
$$

That is, the trace of the first statement is computed and then concatenated with the trace of the rest of the statements. The variable mapping resulting from the translation of each statement is used to compute the trace of its subsequent statements. It should be noted that if the given instance corresponds to an infinite execution of the code, the *toTrace* function will not terminate. Furthermore, if the instance does not correspond to a valid execution of the code, the trace generated by *toTrace* will not be valid.

> ***Example.*** Figure 2-9 shows an example of converting instances to program traces. It gives a trace of the `xorAll` method using the translation previously shown in Figure 2-6. Figure 2-9(a) gives the initial variable and value mappings used in the translation and an instance satisfying the formula generated by the translation. Figure 2-9(b) gives the code of the `xorAll` method. The variable mappings used to translate each statement of the code are given in Figure 2-9(c). Given these values, the *toTrace* function generates the trace of Figure 2-9(d) as a sequence of program statements along with their pre-states.
>
> The *toTrace* operation produces the trace by converting each variable mapping to a program state using the given instance. Since the first statement of the code is an elementary statement, it is included in the trace. Its pre-state is computed using the initial variable mapping. The translation of this statement generates the next variable mapping, `[b -> v1, a[0] -> v2, size -> <v3, v4>, i -> <v5, v6>]`, which is used to extract the trace of the loop: the mapping is converted to a state in which the loop condition (`i < size`) is evaluated. Because `i = 0` and `size = 1`, the condition holds, and thus the trace of the body of the loop is computed. The first statement of the loop is a call to the `xor` method. As shown in the bottom part of Figure 2-9, the trace of the body of this method is computed with respect to a variable mapping that gives the actual mapping of the formal arguments. The

return value of the method (`false`) is substituted for `a[0]` and `i` is incremented. The loop condition is evaluated again. Since it does not hold any more, the execution of the loop terminates.

Abstraction. We compute an initial abstraction of a procedure in which the effects of the call sites are overapproximated using their initial specifications given by the *computeSpec* function. We use the expression *abstract(stmt, varMap, valMap)* to represent the initial abstraction of a program statement *stmt* based on a variable mapping *varMap* and a value mapping *valMap*. The *abstract* function generates a logical formula and a variable mapping:

$$\mathit{abstract}: \mathit{Stmt} \times \mathit{VariableMap} \times \mathit{ValueMap} \rightarrow \mathit{Formula} \times \mathit{VariableMap}$$

The abstraction of an elementary statement is equivalent to its translation. The abstraction of a call site is its initial specification. That is,

$$\mathit{abstract}(\mathit{stmt}: \texttt{ElemStmt}, \mathit{varMap}, \mathit{valMap}) = \mathit{translate}(\mathit{stmt}, \mathit{varMap}, \mathit{valMap})$$
$$\mathit{abstract}(\mathit{stmt}: \texttt{CallStmt}, \mathit{varMap}, \mathit{valMap}) = \mathit{computeSpec}(\mathit{stmt}, \mathit{varMap}, \mathit{valMap})$$

Intuitively, loops and conditionals are abstracted by abstracting their bodies: if they contain any call sites in their bodies, those sites are replaced by their initial specifications. Other statements of their bodies are translated using the *translate* function.

Since the specifications of the call sites overapproximate the called procedures, the *abstract* function generates an over-approximation of a given piece of code. That is, any execution allowed by the *translate* function must be allowed by the *abstract* function too. This is formally defined as follows:

$$\begin{array}{l}
(\mathit{translate}(\overrightarrow{\mathit{stmt}}, \mathit{varMap}, \mathit{valMap}) = (f_t, \mathit{varMap}'_t) \wedge \\
\;\mathit{abstract}(\overrightarrow{\mathit{stmt}}, \mathit{varMap}, \mathit{valMap}) = (f_a, \mathit{varMap}'_a)) \quad \Longrightarrow \\
\quad \forall \mathit{ins}_t \in \mathit{solve}(f_t), \exists \mathit{ins}_a \in \mathit{solve}(f_a) \mid \\
\qquad \mathit{toState}(\mathit{ins}_t, \mathit{varMap}, \mathit{valMap}) = \mathit{toState}(\mathit{ins}_a, \mathit{varMap}, \mathit{valMap}) \wedge \\
\qquad \mathit{toState}(\mathit{ins}_t, \mathit{varMap}'_t, \mathit{valMap}) = \mathit{toState}(\mathit{ins}_a, \mathit{varMap}'_a, \mathit{valMap})
\end{array}$$

```
varMap = [b -> v1, a[0] -> v2, size -> <v3, v4>]
valMap = [false -> 0, true -> 1, 0 -> <0, 0>, 1 -> <0, 1>]
instance = [v1 -> 1, v2 -> 1, v3 -> 0, v4 -> 1, v5 -> 0, v6 -> 0, v7 -> 0, v8 -> 0, v9 -> 1]
```

(a)

void xorAll(**boolean** b,	mappings before translating stmts:	pre-states of executed stmts:	executed stmts:
boolean[] a, **int** size) {			
int i = 0;	varMap=[b->v1, a[0]->v2, size-><v3,v4>]	s=[b-> true, a[0]-> true, size -> 1]	**int** i = 0;
while (i < size) {	varMap=[..., i -> <v5, v6>]		
a[i] = xor(b, a[i]);	varMap=[..., xor$return -> v7]	s = [..., i-> 0, xor$return -> false]	a[i] = xor$return;
i = i + 1;	varMap=[..., , a[0] -> v7]	s = [..., a[0] -> false]	i = i + 1;
}	varMap=[..., i -> <v8, v9>]	s = [..., i -> 1]	exit;
}			
boolean xor(**boolean** x,			
boolean y) {			
if (x == y)			
return false;	varMap=[x -> v1, y -> v2]	s = [x -> true, y -> true]	xor$return = **false**;
else			
return true;			
}	varMap=[..., xor$return -> v7]		
(b)	(c)	(d)	

Figure 2-9: An example of extracting traces from instances: (a) initial variable and value mappings and the instance, (b) the `xorAll` method, (c) variable mappings used to compute the trace of each statement, and (d) the generated trace.

```
varMap = [b -> v1, a[0] -> v2, size -> <v3, v4>]
valMap = [false -> 0, true -> 1, 0 -> <0, 0>, 1 -> <0, 1>]
```

(a)

```
void xorAll(boolean b,
  boolean[] a, int size) {
 int i = 0;
 if (i < size) {
   a[i] = xor(b, a[i]);
   i = i + 1;

   assume (i >= size);

 }
}

boolean xor(boolean x,
   boolean y) {
 if (x == y)
   return false;
 else
   return true;
}
```

(b)

```
mapping after abstracting a stmt:

varMap=[..., i -> <v5, v6>]

varMap=[..., a[0] -> v7]
varMap=[..., i -> <v8, v9>]

varMap=[...]
```

(c)

```
(¬v5 ∧ ¬v6) ∧
((¬v5 ∧ v3) ∨ (v5 ∧ v3 ∧ ¬v6 ∧ v4) ∨ (¬v5 ∧ ¬v3 ∧ ¬v6 ∧ v4)) ⟹ (
  (true) ∧
  ((¬v8 ∧ v9 ∧ ¬v5 ∧ ¬  v6) ∨ (v8 ∧ ¬v9 ∧ ¬v5 ∧ v6) ∨ (v8 ∧ v9 ∧ v5 ∧ ¬v6)∨
   (¬v8 ∧ ¬v9 ∧ v5 ∧ v6)) ∧
  ((v8 ∧ ¬v3) ∨ (v8 ∧ v3 ∧ v9) ∨ (v8 ∧ v3 ∧ ¬v9 ∧ ¬v4) ∨ (¬v8 ∧ ¬v3 ∧ v9)∨
   (¬v8 ∧ ¬v3 ∧ ¬v9 ∧ ¬v4))
)
```

(d)

Figure 2-10: Abstraction example: (a) initial variable and value mappings, (b) the unrolled `xorAll` method, (c) variable mappings after abstracting each statement, and (d) the abstraction of statements as a boolean formula.

That is, any state transition allowed by the *translate* function must correspond to some state transition allowed by the *abstract* function. Therefore, the abstraction of a program contains all executions of the original program and possibly more.

> ***Example.*** Figure 2-10 gives an abstraction of the `xorAll` method. It uses the translation given in Figure 2-6 to translate all the statements except the call to the `xor` method. Instead of inlining the call site, the abstraction uses the initial specification shown previously in Figure 2-7(d) to approximate the behavior of the `xor` method. The specification generates a fresh boolean variable, namely `v7`, for the return value of this method, but leaves it unconstrained, allowing it to take any arbitrary boolean value. Consequently, this abstraction is an overapproximation of the original code.

Extracting call sites. A solution to a formula f that encodes the abstraction of a program represents a valid execution of that program if it assigns valid state transitions to the call sites. To check the validity of an instance *ins*, we define *getCalls* to extract the call sites that are invoked in the program execution corresponding to *ins*. More precisely, the *getCalls* function returns the set of all call statements executed in the trace corresponding to *ins* along with the variable mappings of their pre-states. The definition of *getCalls* is similar to that of *toTrace* except that it works with the *abstract* function rather than *translate*, and only keeps track of the call sites. That is,

```
$\mathit{getCalls} : \mathit{Instance} \times \mathit{Stmt} \times \mathit{VariableMap} \times \mathit{ValueMap} \rightarrow 2^{(\mathit{VariableMap} \times \mathit{CallStmt})}$
$\mathit{getCalls}(\mathit{ins}, \mathit{stmt}_0 : \texttt{ElemStmt}, \mathit{varMap}, \mathit{valMap}) = ()$
$\mathit{getCalls}(\mathit{ins}, \mathit{stmt}_0 : \texttt{CallStmt}, \mathit{varMap}, \mathit{valMap}) = (\mathit{varMap}, \mathit{stmt}_0)$
$\mathit{getCalls}(\mathit{ins}, \texttt{if}\ (\mathit{cond})\ \mathit{stmt}_1\ \texttt{else}\ \mathit{stmt}_2, \mathit{varMap}, \mathit{valMap}) =$
    $\textbf{let}\ \mathit{toState}(\mathit{ins}, \mathit{varMap}, \mathit{valMap}) = s\ \textbf{in}$
    $\textbf{if}\ \mathit{evaluate}(\mathit{cond}, s) = \mathit{true}\ \textbf{then}\ \mathit{getCalls}(\mathit{ins}, \mathit{stmt}_1, \mathit{varMap}, \mathit{valMap})$
                                  $\textbf{else}\ \mathit{getCalls}(\mathit{ins}, \mathit{stmt}_2, \mathit{varMap}, \mathit{valMap})$
$\mathit{getCalls}(\mathit{ins}, \mathit{stmt}_0 = \texttt{while}\ (\mathit{cond})\ \mathit{stmt}, \mathit{varMap}, \mathit{valMap}) =$
    $\textbf{let}\ \mathit{toState}(\mathit{ins}, \mathit{varMap}, \mathit{valMap}) = s\ \textbf{in}$
    $\textbf{if}\ \mathit{evaluate}(\mathit{cond}, s) = \mathit{true}\ \textbf{then}\ \mathit{getCalls}(\mathit{ins}, \mathit{stmt}; \mathit{stmt}_0, \mathit{varMap}, \mathit{valMap})$
                                  $\textbf{else}\ ()$
$\mathit{getCalls}(\mathit{ins}, \mathit{stmt}_0; \overrightarrow{\mathit{stmt}}, \mathit{varMap}, \mathit{valMap}) =$
    $\textbf{let}\ \mathit{abstract}(\mathit{stmt}_0, \mathit{varMap}, \mathit{valMap}) = (f, \mathit{varMap}')\ \textbf{in}$
    $\mathit{getCalls}(\mathit{ins}, \mathit{stmt}_0, \mathit{varMap}, \mathit{valMap}) \cup \mathit{getCalls}(\mathit{ins}, \overrightarrow{\mathit{stmt}}, \mathit{varMap}', \mathit{valMap})$
```

As given by this definition, *getCalls* only returns the call statements that are directly called by the given piece of code; it does not return the nested calls.

> ***Example.*** Figure 2-11 shows an example of extracting calls from an instance. It gives the call sites invoked in an execution of the abstract `xorAll` method. The variable and value mappings of this figure are the initial mappings used in Figure 2-10 to abstract `xorAll`. The instance is a solution to the formula generated by the abstraction. Given these values, the *getCalls* function generates the set given in Figure 2-11, which contains only one pair of variable mapping and call statement. This implies that the trace corresponding to this instance contains only one call to the `xor` method, and the call site has been abstracted using the extracted variable mapping.

```
varMap = [b -> v1, a[0] -> v2, size -> <v3, v4>]
valMap = [false -> 0, true -> 1, 0 -> <0, 0>, 1 -> <0, 1>]
instance = [v1 -> 1, v2 -> 1, v3 -> 0, v4 -> 1, v5 -> 0,
            v6 -> 0, v7 -> 0, v8 -> 0, v9 -> 1]

getCalls(instance, xorAll, varMap, valMap) =
    {([b -> v1, a[0] -> v2, size -> <v3, v4>, i -> <v5, v6>], xor(b, a[i]))}
```

Figure 2-11: An example of extracting calls from an instance.

2.5 Input Functions Properties

Translation properties. A *translate* function is *semantics-preserving* if and only if all the variable mappings it generates are valid, and it is both sound and complete. That is, if $translate(\overrightarrow{stmt}, varMap, valMap) = (f, varMap')$, then the following two rules hold:

$\texttt{completeness:}\ \forall(s, s') \in [\![\overrightarrow{stmt}]\!] \mid solve(f, ins \cup ins') \neq \emptyset\ \texttt{where}$
$ins = toInstance(s, varMap, valMap)\ \textbf{and}\ ins' = toInstance(s', varMap', valMap)$

$\texttt{soundness:}\ \forall ins \in solve(f) \mid (s, s') \in [\![\overrightarrow{stmt}]\!]\ \texttt{where}$
$s = toState(ins, varMap, valMap)\ \textbf{and}\ s' = toState(ins, varMap', valMap)$

That is, (1) any execution of $\overrightarrow{stmt}$ corresponds to some solution to f (completeness), and (2) any solution to f corresponds to some valid execution of $\overrightarrow{stmt}$ (soundness).

We require the *translate* function to return a finite formula. Depending on the translation method, this may require bounding variable domains, heap size, execution length, and/or recursive calls. Therefore, the translation may not be semantics-preserving. If the translation is not complete, i.e. some executions of the translated program do not appear as solutions to the generated formula, the analysis might miss some bugs. On the other hand, if the translation is not sound, i.e. some solutions to the generated formula do not encode valid executions of the translated program, the analysis might report false counterexamples. Thus, the choice of the translation technique will affect the results of the analysis.

The translation example given in Figure 2-6 is sound but not complete. It is not complete because the loop is unrolled once. Thus, no execution in which a loop is iterated more than once will satisfy the formulas generated by the translation. However, it is sound because after unrolling the loop, the translation of all statements preserve their semantics. Thus, any solution to the generated formulas is a valid execution of the code.

Solver Properties. The choice of solver also affects the results of our analysis. We require the solver to be *sound*, meaning that if it returns a solution to the formula, the formula evaluates to true under the mapping defined by that solution. However, the solver does not have to be *complete*. A solver is complete if it guarantees to find a solution satisfying the given formula whenever one exists. If the solver used in our analysis is not complete, meaning that it may miss some solutions, our analysis can miss some bugs. Therefore, the analysis will not be complete. Furthermore, if the solver is not guaranteed to terminate on all inputs, our analysis does not necessarily terminate either.

Initial Specification Properties. We require the initial specifications of a procedure call to under-specify the effects of that call site. This is formalized as follows:

$$
\begin{aligned}
&(\mathit{translate}(\overrightarrow{\mathit{stmt}}, \mathit{varMap}, \mathit{valMap}) = (f_t, \mathit{varMap}'_t) \wedge \\
&\ \mathit{computeSpec}(\overrightarrow{\mathit{stmt}}, \mathit{varMap}, \mathit{valMap}) = (f_s, \mathit{varMap}'_s)) \quad \Longrightarrow \\
&\quad \forall \mathit{ins}_t \in \mathit{solve}(f_t), \exists \mathit{ins}_s \in \mathit{solve}(f_s) \mid \\
&\qquad \mathit{toState}(\mathit{ins}_t, \mathit{varMap}, \mathit{valMap}) = \mathit{toState}(\mathit{ins}_s, \mathit{varMap}, \mathit{valMap}) \wedge \\
&\qquad \mathit{toState}(\mathit{ins}_t, \mathit{varMap}'_t, \mathit{valMap}) = \mathit{toState}(\mathit{ins}_s, \mathit{varMap}'_s, \mathit{valMap})
\end{aligned}
$$

That is, any state transition allowed by *translate* must be allowed by *computeSpec* too. Therefore, the initial specification of a procedure call accounts for all of its possible executions.

2.6 Algorithm

Our analysis method is formalized in the *abstractAnalyze* algorithm of Figure 2-12. This algorithm describes the analysis in terms of the above functions. It takes a procedure *proc* selected by the user for checking, a logical formula *property* representing the property to check, a variable mapping *varMap*, and a value mapping *valMap* to use in the abstraction. The result of the analysis is either `NoCounterexample`, indicating that the property holds in the program, or `Counterexample(t)`, indicating that `t` is a counterexample: a trace in *proc* that violates *property*.

The analysis starts by abstracting all procedures called in the analyzed procedure *proc* (Line 1). The resulting formula is then conjoined with the negation of *property* (Line 2). Therefore, any solution to this formula will be a counterexample to the property. The resulting formula is denoted by the variable f. Throughout the algorithm, this variable represents an abstraction of the code that needs to be checked for a counterexample.

In the solving phase (Line 4), a solver is used to find the instances satisfying f. If no such instance exists (Lines 5 - 6), it means that the abstract program contains no counterexamples. Since the abstraction is an over-approximation of the original program, the original program contains no counterexamples either, and thus, the analysis terminates (Line 6).

Otherwise, an instance *ins* is arbitrarily chosen from the set of solutions to f (Line 7). This instance represents a counterexample in the abstract program which is then checked for validity with respect to the original program (Line 8). It should be noted that the choice of the instance might affect the number of refinements that are needed to check the property. However, it does not affect the analysis result, i.e. whether the property is validated or not.

The validity check phase is performed by the *checkValidity* function (Lines 15 - 28). It checks if an instance *ins* is a valid execution of a procedure *proc* using a variable mapping *varMap* and a value mapping *valMap*. Since the only abstract statements are procedure calls, the *checkValidity* function only needs to check if *ins* is consistent with the call sites of *proc*. Variable *calls* (Line 15) denotes the set of all call sites executed in the program trace corresponding to *ins* along with the variable mappings of their pre-states. These call sites will be analyzed to check the validity of *ins* (Lines 16 - 28).

The *checkValidity* function checks the consistency of *ins* with each called procedure

```
  datatype AnalysisResult = NoCounterexample + Counterexample(Trace)
  datatype ValidityResult = Valid(Instance) + Invalid(Formula)

function $abstractAnalyze$(Procedure $proc$, Formula $property$,
          VariableMap $varMap$, ValueMap $valMap$): AnalysisResult {
    $(f, varMap') = abstract(proc, varMap, valMap)$          ($abstraction$)
    $f = f \wedge \neg property$
    while true {
        $instances = solve(f)$                               ($solving$)
        if $instances = \emptyset$
            return NoCounterexample
        $ins$ = choose($instances$)
        $res = checkValidity(proc, ins, varMap, valMap)$     ($validity\ check$)
        if $res$ = Valid($ins'$)
            $trace = toTrace(ins', proc, varMap, valMap)$
            return Counterexample($trace$)
        if $res$ = Invalid($spec$)                           ($refinement$)
            $f = f \wedge spec$
    }
}

function $checkValidity$(Procedure $proc$, Instance $ins$,
          VariableMap $varMap$, ValueMap $valMap$): ValidityResult {
    $calls = getCalls(ins, proc, varMap, valMap)$
    foreach $(varMap_i, stmt_i) \in calls$
        $proc_i = callee(stmt_i)$
        $(f_i, varMap'_i) = abstract(proc_i, varMap_i, valMap)$
        $instances' = solve(f_i, ins)$
        if $instances' = \emptyset$
            $spec_i = prove(f_i, ins)$
            return Invalid($spec_i$)
        else
            $ins'$ = choose($instances'$)
            $ins = ins \cup ins'$
            $calls = calls \cup getCalls(ins', proc_i, varMap_i, valMap)$
    }
    return Valid($ins$)
}
```

Figure 2-12: Analysis by procedure abstraction.

until either an inconsistent call is found or it is shown to be consistent with all of them. As our abstraction is based on the procedure call hierarchy of the code, the check for validity is done hierarchically. That is, when a call to a procedure $proc_i$ is checked, all of its callees are abstracted. This is why *checkValidity* uses the abstraction of $proc_i$ rather than its translation (Line 18). The abstraction of $proc_i$ generates a formula denoted by the variable f_i (Line 18). If *ins* is not consistent with $proc_i$, that is, if f_i does not have a solution that preserves *ins*, the specification of $proc_i$ should be refined (Lines 20 - 22). In this case, the solver returns an unsatisfiability proof, represented by $spec_i$, which is still unsatisfiable with respect to *ins* (Line 21). This proof is a partial specification for $proc_i$ that rules out the current counterexample *ins*, and possibly more. Line 22 returns $spec_i$ as an inferred specification.

On the other hand, if *ins* is consistent with the semantics of the procedure $proc_i$, it means that some execution of $proc_i$ conforms to the state transition that *ins* has assigned to the call to $proc_i$. All such executions are included in *instances'*. One of those instances, denoted by *ins'*, is chosen (Line 24) to augment the current counterexample *ins* to include the execution within $proc_i$ as well (Line 25). Furthermore, since the call sites of $proc_i$ were abstract when *ins* was checked for consistency with $proc_i$, those call sites should be analyzed too. Therefore, the procedures called in *ins'* are added to the set *calls* (Line 26) to be checked later. If the current counterexample *ins* is consistent with all the call sites needed to be checked, this counterexample is valid and is returned (Line 28).

If the result of the validity check is valid, the *abstractAnalyze* function uses *toTrace* to map the augmented instance (*ins'*) back to an execution in the original program (Line 10). The result is returned as a counterexample (Line 11).

Otherwise, Line 13 conjoins the specification *spec* returned by *checkValidity* with the current formula f to guarantee that *ins* will never be found by the solver again. After this refinement, the solving phase starts over.

It should be noted that *abstractAnalyze* is just an outline of the process. There are several opportunities for optimization in an actual implementation. We explain some of them in Chapters 3 to 6.

Example. We illustrate this algorithm using the xorAll example given in Figure 2-13(a). The given property specifies that if both b and a[0] are true, the method modifies a[0] to be false (a'[0] denotes the final value of a[0]). The initial abstraction of this method was previously given in Figure 2-10. Figure 2-13(b) gives the initial variable and value mappings used in the abstraction along with the final variable mapping that the abstraction generates. These values are used to translate the given property to the boolean formula given in Figure 2-13(c). The property is negated and conjoined with the formulas encoding the abstraction. Solving the resulting formula produces the instance given in Figure 2-13(d). It represents a case where although both b and a[0] are true, the final value of a[0] is not false, it is true instead. Therefore, although it is a counterexample to the property, it is not valid in the original code.

The validity of this instance is checked by checking the original code of the call sites it invokes. The trace corresponding to this instance invokes only one method call: the call to the xor method in the first iteration of the loop. This call site and its pre-state variable mapping is extracted using the *getCalls* function as given in Figure 2-13(e). To check the validity of this instance, the body of the xor method is translated to a boolean formula with respect to the extracted pre-state variable mapping. This translation is shown in Figure 2-13(f). Solving the generated formula with respect to the found instance results in no solutions, implying that the instance is invalid. The solver generates the proof of unsatisfiability given in Figure 2-13(g), which specifies that if the two arguments of the xor method are equal, the return value is false. This formula is conjoined with the abstraction of xorAll and solved again against the property. In this example, no more solutions are found, and the property has been validated.

```
// ((b = true) and (a[0] = true)) =>
//     (a'[0] = false)
void xorAll(boolean b, boolean[] a, int size) {
  int i = 0;
  while (i < size) {
    a[i] = xor(b, a[i]);
    i = i + 1;
  }
}

boolean xor(boolean x, boolean y) {
  if (x == y) return false;
  else return true;
}
```

(a)

```
varMap = [b -> v1, a[0] -> v2,
          size -> <v3, v4>]
valMap = [false -> 0, true -> 1,
        0 -> <0, 0>, 1 -> <0, 1>]

varMap' = [b -> v1, a[0] -> v7,
 size -> <v3, v4>, i -> <v8, v9>]
```

(b)

$\mathtt{v1} \wedge \mathtt{v2} \implies \neg\mathtt{v7}$

(c)

```
instance = [v1 -> 1, v2 -> 1,
v3-> 0, v4-> 1, v5 -> 0, v6 -> 0,
v7 -> 1, v8 -> 0, v9 -> 1]
```

(d)

```
getCalls(instance, ..) = { ([b -> v1, a[0] -> v2, size -> <v3, v4>, i -> <v5, v6>],
                          xor(b, a[i])) }
```

(e)

```
varMap= [b -> v1, a[0] -> v2, size -> <v3, v4>,
         i -> <v5, v6>, xor$return -> v7]
```

$(((\neg\mathtt{v1}\wedge\neg\mathtt{v2})\vee(\mathtt{v1}\wedge\mathtt{v2})) \implies \neg\mathtt{v7})$
$\wedge(((\neg\mathtt{v1}\wedge\mathtt{v2})\vee(\mathtt{v1}\wedge\neg\mathtt{v2})) \implies \mathtt{v7})$

(f)

$(\mathtt{v1} \wedge \mathtt{v2}) \implies \neg\mathtt{v7}$

(g)

Figure 2-13: An illustration of the analysis: (a) the xorAll method and its property of interest, (b) initial and final mappings involved in the abstraction, (c) the encoded property, (d) the found instance, (e) the call site invoked by the instance, and its pre-state variable mapping, (f) the translation of the call to the xor method, (g) inferred specification.

```
  datatype AnalysisResult = NoCounterexample + Counterexample(Trace)

function $inlinedAnalyze$(Procedure $proc$, Formula $property$,
           VariableMap $varMap$, ValueMap $valMap$): AnalysisResult {
    $(g, varMap') = translate(proc, varMap, valMap)$
    $g = g \wedge \neg property$
    $instances = solve(g)$
    if $instances = \emptyset$
       return NoCounterexample
    $ins =$ choose($instances$)
    $trace = toTrace(ins, proc, varMap, valMap)$
    return Counterexample($trace$)
}
```

Figure 2-14: Analysis without abstraction.

2.7 Properties

The termination, completeness, and soundness properties of the described framework depends on the actual translation, solving, and proof generating techniques used to instantiate it. Therefore, in this section, we analyze these properties by comparing our framework (the *abstractAnalyze* algorithm) with a similar analysis technique that does not abstract procedures, namely the *inlinedAnalyze* algorithm given in Figure 2-14.

2.7.1 Termination

The *abstractAnalyze* algorithm contains two iteration points: a refine-solve cycle (Lines 3 - 14), and a validity check cycle (Lines 16 - 27). The syntax of Figure 2-2 allows analyzed programs to contain executions with infinite loops and recursive procedure calls. The analysis of such programs can loop forever because the set of call sites to check can grow without bound. If the program execution corresponding to a found counterexample calls a procedure p infinitely (either in a loop or a recursive chain of calls), all those call sites have to be checked for consistency with the counterexample. Therefore, checking the validity of the counterexample will not terminate.

Furthermore, even if the analyzed program does not contain infinite executions, the axioms defined in this chapter allow the *prove* function to rule out spurious counterexamples one by one by returning the negation of a counterexample as the proof of unsatisfiability in each iteration. Since the abstract program can have infinitely many spurious executions that violate a property, the refine-solve cycle may not terminate either. Thus, in general, the analysis is not guaranteed to terminate.

However, as explained in Chapters 3 to 6 our instantiation of this framework translates the code by finitizing the loops and recursive calls and analyzes it with respect to a finite heap. Therefore, it guarantees that a finite number of counterexamples exist, and each of them represents a finite execution of the code. That is, each counterexample contains only a finite number of call sites to check. Therefore, our instantiation of the framework is guaranteed to terminate.

2.7.2 Completeness

An error detecting analysis is *complete* if and only if whenever there exists a counterexample to a given property, the analysis can find it. As discussed before, our technique is not guaranteed to terminate. Therefore, in general, our analysis is not complete.

However, if the basic operations and the analyzed program are chosen so that the analysis is guaranteed to terminate, we argue that our *abstractAnalyze* algorithm is as complete as the *inlinedAnalyze* algorithm. Therefore, the specification refinement idea does not introduce incompleteness in terminating analyses. The proof is as follows.

Lemma 1. *In each iteration of the abstractAnalyze algorithm, the solved formula f is weaker than the formula g solved in inlinedAnalyze.*

Proof. The proof is by induction. The formula f is initially equal to the formula generated by *abstract(proc, varMap, valMap)* which is weaker than the formula generated by *translate(proc, varMap, valMap)* because of the over-approximation property of *abstract.* Therefore, for a given *property*, the formula f solved in the first iteration of *abstractAnalyze* is weaker than the formula g solved in *inlinedAnalyze*. Thus, the base of the induction is true.

Let f^j denote the formula f solved in the j^{th} iteration of *abstractAnalyze*. We show that if f^j is weaker than g, then f^{j+1} is also weaker than g. The proof is as follows:

$$\begin{aligned}
&f^{j+1} = f^j \wedge \mathit{spec} && \text{(Line 13)}\\
&\mathit{spec} = \mathit{prov}(f_i, \mathit{ins}) && \text{(Line 21)}\\
&(f_i, \mathit{varMap}'_i) = \mathit{abstract}(\mathit{callee}(\mathit{stmt}_i), \mathit{varMap}_i, \mathit{valMap}) && \text{(Line 18)}\\
&(\mathit{varMap}_i, \mathit{stmt}_i) \in \mathit{calls} && \text{(Line 16)}\\
&\mathit{calls} = \mathit{getCalls}(\mathit{ins}, \mathit{proc}, \mathit{varMap}, \mathit{valMap}) && \text{(Line 15)}
\end{aligned}$$

By the definition of *prove*, the formula *spec* is weaker than f_i, which is weaker than the formula generated by $\mathit{translate}(\mathit{callee}(\mathit{stmt}_i), \mathit{varMap}_i, \mathit{valMap})$ because of the over-approximation property of *abstract*. Since $\mathit{callee}(\mathit{stmt}_i)$ is reachable from *proc* (by Line 15), the formula generated by $\mathit{translate}(\mathit{callee}(\mathit{stmt}_i), \mathit{varMap}_i, \mathit{valMap})$ is weaker than the one generated by $\mathit{translate}(\mathit{proc}, \mathit{varMap}, \mathit{valMap})$. Therefore, *spec* is weaker than g. Since both f^j and *spec* are weaker than g, the formula f^{j+1} is weaker than g too. Thus, the given invariant holds. □

Theorem 1. *If inlinedAnalyze finds a counterexample and abstractAnalyze does not loop forever, it finds a counterexample too.*

Proof. Assume to the contrary that *inlinedAnalyze* returns a counterexample, but *abstractAnalyze* terminates with no counterexamples. That is, *abstractAnalyze* terminates in Line 6 implying that $\mathit{solve}(f) = \emptyset$. However, by Lemma 1, the formula f solved in each iteration of *abstractAnalyze* is weaker than the formula g solved in *inlinedAnalyze*. Therefore, if $\mathit{solve}(f) = \emptyset$ then, $\mathit{solve}(g) = \emptyset$. That is, *inlinedAnalyze* does not find any counterexamples either, contradicting the assumption. □

2.7.3 Soundness

An error detecting analysis is *sound* if and only if all the counterexamples it returns are feasible executions of the analyzed code that violate the analyzed property. We show that our specification refinement idea does not introduce unsoundness. That is, the following theorem holds:

Theorem 2. *If all counterexamples returned by inlinedAnalyze are feasible executions of the analyzed code, all counterexamples returned by abstractAnalyze are also feasible.*

Proof. Let *abstractAnalyze(proc, property, varMap, valMap)* generate a trace *t* corresponding to an instance *ins*. According to the *abstractAnalyze* algorithm, all procedures called in *t* have been checked for validity before *t* is returned. Thus, *ins* satisfies all the formulas generated by *translate(proc, property, varMap, valMap)* that encode those program statements that are executed in *t*. All other formulas generated by *translate(proc, property, varMap, valMap)* encode program statements unreachable by *t* and thus vacuously evaluate to true under the instance *ins*. Thus, *ins* satisfies all the formulas generated by *translate(proc, property, varMap, valMap)*, and its corresponding trace, *t*, can be returned by *inlinedAnalyze*, too. Since all counterexamples returned by *inlinedAnalyze* are feasible executions, *t* is also feasible. Therefore, all counterexamples returned by *abstractAnalyze* are feasible too. □

Chapter 3

Context

In this chapter, we describe the inputs that the user provides to our analysis technique. Our technique checks a given program against a property, both provided by the user. The program is assumed to be written in a subset of Java, and the property should be expressed in the Alloy modeling language. The analysis is performed with respect to a bounded domain where the bounds are parameters also provided by the user. The output of the analysis is either a valid counterexample, indicating that the code does not satisfy the property, or a guarantee that the property holds within the analyzed bounds; though nothing is guaranteed beyond those bounds.

3.1 Programming Language: Java

We focus on checking object-oriented programs. Our tool, Karun, currently supports a subset of Java that includes basic statements and inheritance. It does not include arrays, reflection, exceptions, concurrency, or complex numerical expressions (only integer comparisons, additions, and subtractions are supported). Figure 3-1 gives a grammar for supported program statements.

```
Stmt     ::= Var = PExpr
           | PExpr.Field = PExpr
           | Var = new Class(PExpr*)
           | Proc(PExpr*)
           | Var = Proc(PExpr*)
           | return PExpr
           | if (Cond) Stmt [else Stmt]
           | while (Cond) Stmt
           | Stmt; Stmt
PExpr    ::= IntConst | PConst | Var[.Field]*
           | PExpr + PExpr | PExpr - PExpr
IntConst::= 0 | 1 | -1 | 2 | -2 | ...
PConst   ::= null | true | false
Cond     ::= Var[.Field]* == PExpr | Var[.Field]* != PExpr
           | PExpr < PExpr | PExpr > PExpr | PExpr instanceof Class
           | Cond && Cond | Cond || Cond
```

Figure 3-1: Program statements.

3.2 Specification Language: Alloy

In order to check a program, our technique requires the user to provide a property which is a top-level specification that (partially) specifies the behavior of a procedure in the given program. We particularly target *data structure properties* of the code, i.e., properties that constrain configurations of the objects in the heap.

We assume that the given property is expressed in Alloy [30], a first order relational logic that includes transitive closure, making it well suited for expressing complex data structure properties succinctly. Our tool supports all of Alloy. Thus, any well-formed Alloy formula is accepted as a property.

A property specifies the behavior of a procedure p as (partial) pre- and post-conditions for p expressed in terms of the classes and fields declared in the program, and formal parameters and the return value of p. We assume that the values in the pre- and post-states are referenced by unprimed and primed identifiers respectively.

Figure 3-2 gives the syntax of the Alloy logic. Although user-provided properties can be any Alloy formulas, the code abstractions automatically generated by our tool use only a subset of the Alloy language. The details are explained in Chapters 4 through 6.

```
AlloyModel::= TypeDecl* RelDecl* Formula
TypeDecl  ::= sig TypeId | sig TypeId extends TypeId
RelDecl   ::= RelId: Expr[ -> Expr]*
Expr      ::= TypeId | RelId
            | Int | univ | iden | none
            | Expr + Expr                            union
            | Expr & Expr                            intersection
            | Expr - Expr                            difference
            | Expr.Expr                              composition
            | Expr -> Expr                           product
            | Expr ++ Expr                           override
            | ~Expr                                  transpose
            | *Expr | ^Expr                          closure
            | {var: Expr | Formula}                  set comprehension
            | if Formula then Expr else Expr         conditional
            | Int(IntExpr)                           integer
IntExpr   ::= intConst | int(Expr)
            | size(Expr)                             set cardinality
            | if Formula then IntExpr else IntExpr   conditional
            | intExpr + intExpr                      addition
            | intExpr - intExpr                      subtraction
IntConst  ::= 0 | 1 | -1 | 2 | -2 | ...
Formula   ::= Expr in Expr                           subset
            | Expr = Expr                            equality
            | Formula and Formula                    conjunction
            | Formula or Formula                     disjunction
            | not Formula                            negation
            | Formula => Formula                     implication
            | Formula <=> Formula                    bi-implication
            | all var:Expr | Formula                 universal
            | some var: Expr | Formula               existential
            | one var: Expr | Formula                singleton
            | no Expr | some Expr | one Expr         cardinality
            |. IntFormula                            integer
IntFormula::= IntExpr = IntExpr
            | IntExpr < IntExpr
            | IntExpr > IntExpr
TypeId    ::= Identifier
RelId     ::= Identifier
```

Figure 3-2: Syntax of the Alloy logic.

3.2.1 Declarations

Alloy is a strongly typed language that assumes a universe of atoms partitioned into disjoint sets, each of which is associated with a basic type. A basic type T is declared as **sig** T denoting a set of atoms of type T.

Subtypes in Alloy are represented as subsets and are declared using the **extends** keyword. The subtypes of a type T are mutually disjoint, i.e. they share no atoms.

The value of any expression in Alloy is always a relation – that is a set of tuples of atoms. A relation declaration denotes the relation's type and arity; any finite arity is allowed. For example, if E_1, E_2 and E_3 are three expressions, then a relation of type E_1 `->` E_2 `->` E_3 is a set of tuples whose first element is chosen from the set representing E_1, second one is chosen from the set representing E_2 and third one is chosen from the set representing E_3. The arity of such relation is the sum of the arities of E_1, E_2, and E_3.

Sets of atoms are expressed as unary relations, i.e. relations whose tuples have only one column. Scalars are expressed as singleton unary relations, i.e. relations containing only one tuple with only one column.

3.2.2 Expressions

Alloy expressions may reference type identifiers and relation identifiers. In addition to that, there are four constants in the Alloy language: **Int** is the only built-in type which represents a set of atoms of type integer. The actual integer value of these atoms will be determined during the analysis of the Alloy model. The **univ** relation denotes a relation containing all the atoms in the universe (a union of all types). The **iden** relation denotes an identity relation defined over **univ**, and the **none** relation represents an empty relation.

Relations are combined with a variety of operators to form expressions. These operators are either set operators or relational operators. The standard set operators, `+` for union, `&` for intersection, and `-` for difference, combine two relations of the same type, viewed as sets of tuples, according to their standard definitions.

The main relational operator is the dot operator for join (relational composition). The composition of relations r (arity n) and s (arity m) is defined as follows:

$$\begin{aligned} r.s = \{\langle r_1, .., r_{n-1}, s_2, .., s_m\rangle \mid \\ \langle r_1, .., r_n\rangle \in r \ \wedge \langle s_1, .., s_m\rangle \in s \ \wedge \ (r_n = s_1)\} \end{aligned}$$

If r is a unary relation ($n = 1$) and s is a binary relation ($m = 2$), the expression $r.s$ gives the standard relational image of r under s. If r is a binary relation ($n = 2$) and s is a unary relation ($m = 1$), the expression $r.s$ gives the set of atoms of the domain of r whose mapping under r equals some atom in s.

Other relational operators are product and override. The product operator r -> s gives the cross product of the two relations r and s. The override operator is defined as follows:

$$\begin{aligned} r \texttt{ ++ } s = \{\langle a_1, .., a_n\rangle \mid \langle a_1, .., a_n\rangle \in s \vee \\ (\langle a_1, .., a_n\rangle \in r \ \wedge \ (r_1 \notin \mathit{domain}(s)))\} \end{aligned}$$

That is, (r ++ s) contains all the tuples of s and those tuples of r whose first atom is not mapped to anything by s.

There are also two unary operators: transpose and closure. The transpose of a relation r, denoted by ~r, gives a new relation by reversing the order of atoms in each tuple of r. The closure operator ^ takes a homogeneous binary relation $r : T$ -> T and gives the transitive closure of r defined as follows:

$$\hat{\ }r = r \ + \ r.r \ + \ r.r.r \ + \ ...$$

The operator $*$ also computes a transitive closure, but its result is always reflexive. That is,

$$* \ r = \mathit{iden} + r \ + \ r.r \ + \ r.r.r \ + \ ...$$

The expression $x.{*}r$ succinctly represents all the elements reachable from x via the r relation.

In addition to the above operators, an expression may be formed based on the evaluation of some formula. A set comprehension expression $\{v : e \mid f(v)\}$ defines a set of atoms that belong to the expression e and satisfy the formula $f(v)$. A conditional expression specifies a relation based on the truth value of the branching formula. That is,

$$\textbf{if } f \textbf{ then } e_1 \textbf{ else } e_2 = \begin{cases} e_1 & \texttt{in case } f \texttt{ is true} \\ e_2 & \texttt{otherwise} \end{cases}$$

For an integer expression i, the expression $\mathbf{Int}(i)$ denotes the integer atom corresponding to the integer value of i. An integer expression can be a constant, the value of an integer atom, denoted by $\mathbf{int}(e)$, a set cardinality, an integer conditional expression, or addition and subtraction expressions.

3.2.3 Formulas

Elementary formulas are formed from the subset operator **in** and the equivalence operator =. For two expressions p and q, the formula p **in** q is true when every tuple in p is also in q and the formula $p = q$ is true when p **in** q and q **in** p are both true.

Larger formulas can be obtained using the standard logical connectives: conjunction, disjunction, negation, and implication. First order quantifiers are also allowed. For example, **one** $x : e \mid F$ is true if and only if the formula F holds under exactly one binding of the variable x to a scalar from the set denoted by e.

For a quantifier Q and an expression e, the formula $Q\ e$ constrains the cardinality of the relation representing e. The formulas **no** e, **some** e, and **one** e respectively denote that e is empty, non-empty, and a singleton relation.

Formulas may also be formed from integer expressions. Test for equality, less than, and greater than are supported by the language.

3.2.4 Auxiliary Constraints

The following two functions that are provided by the Alloy engine will be extensively used in the translation of Java programs to Alloy.

- **Functionals:** A relation $r : T_1$ -> T_2 is functional if and only if r maps each atom of T_1 to at most one atom in T_2. The **functional** constraint is defined as follows:

  $$\textit{functional}(r : T_1 \texttt{ -> } T_2) \ \{ \ \textbf{all} \ x : T_1 \mid ((\textbf{no} \ x.r) \ \textbf{or} \ (\textbf{one} \ x.r)) \ \}$$

- **Total orders:** A homogeneous relation $r : T$ -> T is a total order if and only if all atoms of T are reachable from some atom *first* by traversing r. Furthermore, every atom in T, except some *last* atom, has exactly one mapping under r. More precisely, the **totalOrder** constraint is defined as follows:

  $$\begin{array}{l} \textit{totalOrder}(r : T \texttt{ -> } T) \ \{ \\ \qquad (\textbf{one} \ \textit{first} : T \mid \textit{first}. * r = T) \quad \textbf{and} \\ \qquad (\textbf{one} \ \textit{last} : T \mid ((\textbf{all} \ x : (T - \textit{last}) \mid \textbf{one} \ x.r) \ \textbf{and} \ (\textbf{no} \ \textit{last}.r))) \ \} \end{array}$$

3.2.5 Examples

The relational logic underlying Alloy and its built-in transitive closure operator make it well suited for expressing data structure properties of heap-manipulating programs. In this section, we provide sample properties of two kinds of data structures, namely linked lists and binary trees, and illustrate how they can be expressed in Alloy.

Consider the implementation of a singly linked list given below:

```
class List {
  Entry head;
}
class Entry {
  Entry next;
  Data data;
}
class Data {}
```

Adopting a relational view of the heap, each datatype is represented as an Alloy signature, and each field is encoded as a binary function. The head field, for example, is represented by a relation head: List -> Entry that maps each atom of type List to an atom of type Entry. Consequently, dereferencing the field head of a list object l will be equivalent to the relational join l.head.

Using the Alloy transitive closure operator, the set of all entries of a list l can be succinctly specified by the expression l.head.*next, and thus the contents of l is expressed by l.head.*next.data. These expressions become handy in specifying list operations. For example, to encode that a removeAll operation removes all occurrences of a datum d from a list l, it is sufficient to say

```
l.head'.*next'.data' = l.head.*next.data - d
```

That is, the final contents of l are the initial contents of l minus the removed datum d. (We use primed names to denote the values of the relations after an operation.)

Another example is an implementation of a binary tree data structure given below:

```
class Tree {
  Tree left;
  Tree right;
  Tree parent;
  Data data;
}
```

We can constrain the parent field to be consistent with the left and right fields using the following Alloy formula.

```
all x, y: Tree | (x.parent = y) <=> ((y.left = x) or (y.right = x))
```

That is, a tree node y is the parent of a tree x if and only if x is either the left or the right child of y.

Using the union of two relations, the set of all children of a tree t can be succinctly encoded by the t.^(left + right) expression. Thus, we can specify that a tree t is acyclic by the following formula:

```
all t: Tree | not (t in t.^(left + right))
```

That is, no tree node should be included in its set of children.

Furthermore, one can specify that a tree t is balanced using the cardinality operator of Alloy:

```
all n: t.*(left + right) |
     size(n.left.*(left + right)) = size(n.right.*(left + right))
```

That is, for all nodes of the tree, the number of nodes in the left subtree should be equal to the number of nodes in the right subtree.

Although Alloy is expressive enough to specify many complex data structure properties, there are certain properties that cannot be expressed in Alloy. As mentioned in Chapter 1, a property that requires higher-order quantification is an example. To illustrate this, consider a method that takes a linked list and makes it "well-formed" (based on some definition of well-formedness) by updating the next fields of its entries. We can check that the method is correct by analyzing the property wellFormed(next') where next' is the resulting next link in the post-state. However, if we would like to check that the modified list is the closest well-formed list to the original one (based on some definition of distance), we should analyze the following property:

```
wellFormed(next') and
all next0: Entry -> Entry | wellFormed(next0) =>
          (distance(next, next0) >= distance(next, next'))
```

This property, however, requires support for higher-order quantification because of the quantifier over the relation next0. Therefore, it cannot be expressed in Alloy.

3.3 Bounds

An Alloy model can be automatically analyzed with respect to some finite *scope*. The scope defines a bound on the number of atoms of each type. For the integer type, however, the scope is denoted by a bitwidth for integer values rather than the number of integer atoms. For a given bitwidth n, the analysis considers all the numbers from -2^{n-1} to $2^{n-1}-1$ as possible values for integer expressions. Furthermore, the bound on the number of atoms of type **Int** is automatically set to 2^n.

Our program analysis technique generates Alloy models from Java code automatically. As explained in Chapter 5, the types used in the generated Alloy model correspond to the datatypes used in the analyzed program. Therefore, the bounds needed to check the Alloy model represent the maximum number of objects that will be considered for each datatype during the analysis. These bounds are provided by the user and are fixed over the course of the analysis.

Furthermore, the translation technique that we use unrolls loops and recursions before translating the code to Alloy. The maximum number of iterations of the loops and recursive calls is also provided by the user.

Although our analysis checks programs with respect to finitized domains, it considers all possible initial configurations of the heap for initial states. Therefore, the user does not need to provide initial configurations; they are all exhaustively checked automatically. Consequently, if no counterexample is found, it is guaranteed that the property holds within the analyzed domains. However, because of finitizations, it is not a general proof of correctness – thus called *bounded verification.*

Chapter 4

Extracting Initial Specifications

In order to analyze a procedure against a property, we compute an initial abstraction of the procedure in which all call sites are represented by some initial specifications. This chapter describes how the initial specifications of the called procedures are extracted. The technique is static, fully automatic, and does not require any user-provided guidance. It generates syntactic specifications of how a procedure manipulates the objects in the heap.

4.1 Overview

The initial specification of a procedure call need not be accurate; the only requirement is to over-approximate the effects of the corresponding procedure. Therefore, even an empty specification that allows the procedure to change the state of the program arbitrarily is allowed. However, starting with more informative specifications can reduce the number of refinements needed during the analysis. On the other hand, since some procedures are irrelevant to the property being checked, it is not always beneficial to spend considerable time to extract very rich initial specifications.

In this chapter, we describe a lightweight technique for extracting cost-effective initial specifications. Given a procedure, the technique summarizes its behavior by computing a symbolic relationship between the procedure's pre-states and post-states. A summary is a declarative formula expressed in a subset of the Alloy language that does not include quantifiers or set comprehensions. It gives an over-approximation of the procedure's behavior by bounding the final values of accessed fields, the return value, and allocated objects from both above and below by relational expressions.

Upper and lower bounds on relations. Our abstraction technique is a flow-sensitive, context-sensitive instantiation of the abstract interpretation framework [13] in which the abstract domain consists of relational expressions. The pre-state of a procedure is abstracted by representing each type, variable, and field as a relation with some symbolic value. The code is executed symbolically to compute new relational expressions for the values updated by each statement. For each relation, two expressions are computed simultaneously: a lower bound, representing the tuples that occur in *all* executions of the procedure (encoding the procedure's *must* side-effects), and an upper bound, representing the tuples that *possibly* occur in some execution (encoding the procedure's *may* side-effects).

The constraints place no restrictions on aliasing between symbolic names, and thus the results account for all possible aliasing in the pre-state.

Over-approximation of final states. Having computed a lower and an upper bound on the final value of each relation, we construct a summary that over-approximates the set of all possible post-states of the procedure. The summary is an Alloy formula that constrains the final values of all types, variables, and fields to conform to their lower and upper bounds. That is, for each type, variable, or field `x`, the summary contains a constraint of the form

```
(lb in x') and (x' in ub)
```

where `x'` is the relation representing the final value of `x`, and `lb` and `ub` are its lower and upper bounds respectively. This constraint specifies that all tuples of `lb` must be included in `x'` and all tuples of `x'` must be chosen from `ub`.

Syntactic domain of abstract values. The bounds computed by our technique are expressions over the relations that represent the pre-state of the procedure and some fresh relations that represent its allocated objects. These expressions are built and stored *syntactically*. That is, they cannot be evaluated unless the pre-state and allocation relations are bound to specific tuples. Therefore, in order to combine these expressions, we use *syntactic* relational operators. That is, to combine two expressions e_1 and e_2 with a relational operator $\circ$, we use the expression $e_1 \boxed{\circ} e_2$ that gives a syntactic concatenation of the operator $\circ$ with the operands e_1 and e_2.

Representation of allocated objects. The objects allocated by a procedure (directly or indirectly) are represented by fresh unary relations. These relations are not bound to any particular values; they are free variables in the resulting summary. For each datatype, our abstraction technique enumerates the first m objects allocated by the analyzed procedure or its callees, where m is a constant parameter of the analysis. Any further allocations are approximated by a symbolic set of objects whose cardinality is unspecified.

Abstraction of call sites. Call sites of a procedure are abstracted in a context-sensitive way. The context of a call site denotes the variables, fields, and types whose summaries are accurate. That is, their lower and upper bounds are the same. A called procedure may have different summaries at different contexts. However, all calls to a procedure that have the same context share the same summary structure. Therefore, for each calling context of a procedure, we compute a *template summary* that abstracts the behavior of that procedure at that context, using symbolic names for its input arguments, fields, and types. The actual summary of a call site is then computed by instantiating its template summary using actual values of the arguments, fields, and types at the call site.

Abstraction of loops. In order to generate more precise summaries, our analysis exploits the condition of a loop in abstracting the loop body: upon entering a loop, a relational encoding of the loop condition is intersected with the expressions computed for the variables used in that condition. Thus, those tuples that violate the loop condition will be removed. The loop is then abstracted by computing a fixpoint. Then, a relational encoding of the termination condition (negation of the loop condition) is intersected with the final expressions of the condition's variables.

Our analysis is further optimized by applying a pattern-matching step that recognizes the loops iterating over all the elements of a linked data structure, a common pattern in heap-manipulating programs. It generates a more accurate specification for any loop that follows this pattern.

Widening rules. A collection of simplification rules and widenings reduces the size and complexity of the generated expressions, and guarantees that the technique terminates. Two widening rules are designed to concisely summarize the behavior of loops: (1) a union of k relational joins is widened to a transitive closure to soundly approximate the set of objects that are visited during the execution of a loop, and (2) an intersection with the loop condition is dropped from the expressions computed for the variables used in a loop to make them amenable to the previous widening rule. Furthermore, we limit the size of relational expressions by placing a bound n on the number of operators they can contain. Any upper bound exceeding n is widened to the universal relation of the appropriate type. Any lower bound exceeding n is approximated by the empty set.

Applicability to other settings. It should be noted that the summaries that are generated by this technique are valid consequences of the code; they do not depend on finitizations of either heaps or loop executions. Thus, they can be used as general specifications in a variety of settings beyond the context of our analysis framework. Furthermore, although default values are provided for the widening parameters, they can be overridden by user-provided values. This allows users to tune the precision of the generated specifications according to their needs.

4.2 Logic

Our analysis summarizes a procedure's behavior in a subset of the Alloy modeling language. Each class C declared in the analyzed program is encoded by two sets (unary relations): one representing all possible objects of type C, and the other representing all objects of type C that are already allocated. (The latter set is a subset of the former set.) A variable in the program is represented by a singleton set that denotes the value of that variable, and a field of type T declared in a class C

```
Summary    ::= Constr*
Constr     ::= Rel' Op Expr
Rel        ::= Field | Var | TypeId$alloc
Op         ::= ⊆ | ⊇ | =
Expr       ::= Const | none | Rel | TypeId | Int | Allocation
                | Expr + Expr                 union
                | Expr & Expr                 intersection
                | Expr - Expr                 difference
                | Expr.Expr                   composition
                | Expr -> Expr                product
                | Expr ++ Expr                override
                | ^Expr                       transitive closure
                | *Expr                       reflexive transitive closure
Allocation::= New$SymObj | NewSet$SymObj
Field      ::= Identifier
Var        ::= Identifier
Const      ::= Identifier
SymObj     ::= Identifier
TypeId     ::= Identifier
```

Figure 4-1: Syntax of summaries.

is encoded as a functional binary relation of type C -> T that maps each object of type C to at most one object of type T.

Figure 4-1 gives a grammar for the subset of Alloy used in the generated summaries. A summary is a set of constraints, implicitly conjoined, on the post-state relations that represent the final values of a procedure's accessed fields, return value, and allocated objects. The pre- and post-state relations are represented by unprimed and primed names, respectively. For a type T, we use the relation T$alloc to represent the set of allocated objects of type T, and the relation T to represent the set of all possible objects of type T.

A constraint can bound the final values from either above or below. For a relation r and a relational expression e, the subset constraint $r \subseteq e$ expresses that each tuple of r is contained in e. Similarly, the superset constraint $r \supseteq e$ expresses that r contains all the tuples in e. Of course, these constraints are respectively written as (r **in** e) and (e **in** r) in Alloy. However, we use the subset and superset mathematical symbols to improve readability of the examples. We use the equality constraint $r = e$ when both $r \subseteq e$ and $r \supseteq e$ hold.

```
class Graph {
   List nodes;
}
class Node {
   List inNodesList;
   List outNodesList;
   int inDeg;
   int outDeg;
   int visitedInsNum;
}
class List {
   ListEntry head;
}
class ListEntry {
   Node node;
   ListEntry next;
   ListEntry prev;
}
```

Figure 4-2: Declaration part of the example in Chapter 1.

A constraint bounds the final value of a relation by an expression. An expression can be a relation representing a program constant (`Const`), the empty relation (`none`), a relation in the pre-state (representing a formal parameter, a field, or a type allocation), the set of all possible objects of some type (`TypeId`) including the integer type (`Int`), and a single object (`New$SymObj`) or a set of objects (`NewSet$SymObj`) newly allocated by the procedure. All Alloy operators except quantifiers and set comprehensions are also allowed.

4.3 Examples

In this section, we illustrate the summaries that our technique extracts for some small procedures that manipulate the `Graph` data structure previously introduced in the example of Chapter 1. A graph – declared again in Figure 4-2 – is represented by its list of nodes where each node is represented by its adjacency lists of incoming and outgoing edges, its in- and out-degrees, and an auxiliary field `visitedInsNum`. The list of nodes is declared as a doubly linked list.

Figures 4-3 - 4-7 give the summaries for some small procedures. The first two

```
/*
  node' = node ++ (e1 −> e2.node) ++ (e2 −> null)
  nullifyMove$return = e1.(node ++ (e1 −> e2.node) ++ (e2 −> null))
*/
static Node List.nullifyMove(ListEntry e1, ListEntry e2) {
    e1.node = e2.node;
    e2.node = null;
    return e1.node;
}
```

Figure 4-3: Simple update and read statements.

examples illustrate the abstractions of basic program statements, while the last three illustrate the summaries generated for the loops involved in the topological sort example of Chapter 1. We assume that any relation not listed in a summary is asserted to have the same value in the pre- and post-state.

The `nullifyMove` method given in Figure 4-3 shows the abstraction of field updates and navigations. An update to a field is encoded by a relational override which permits a succinct description of exactly which parts of a relation are changed and how. The constraint `node' = node ++ (e1 -> e2.node) ++ (e2 -> null)` encodes that the `node` field of `e1` is set to the `node` field of `e2` in the pre-state, and then the `node` field of `e2` is set to `null` (the override operator associates left to right). The `node` fields of all other objects are unchanged. Navigations are encoded by relational joins. For example, reading the field `node` of an object `e1` is expressed as a relational join of the expression corresponding to `node` and the one corresponding to `e1`. It should be noted that the generated summary specifies the behavior of the method in the general case; it is correct whether or not `e1` and `e2` are aliased: if they are not aliased, the return expression simplifies to `e2.node`, and if they are aliased, the return expression simplifies to `null` (which probably represents an error).

The `insert` method given in Figure 4-4 shows the abstraction of object allocation and branch statements. It takes a node `n`, allocates a new list entry to store `n`, and inserts it at the beginning of the receiver list, `this`. The allocated entry is represented by the symbolic name `New$ListEntry`. The updates made to the `node`, `head`, and `next` fields are straightforward, and their final summaries are accurate. The update

```
/*
  ListEntry$alloc' = ListEntry$alloc + New$ListEntry
  node' = node ++ (New$ListEntry -> n)
  head' = head ++ (this -> New$ListEntry)
  next' = next ++ (New$ListEntry -> this.head)
  prev' ⊆ (prev ++ New$ListEntry -> null) + (this.head -> New$ListEntry)
  prev' ⊇ (prev ++ New$ListEntry -> null) - (this.head -> ListEntry)
*/
void List.insert(Node n) {
    ListEntry e = new ListEntry(n);
    ListEntry tmp = this.head;
    this.head = e;
    e.next = tmp;
    if (tmp != null)
        tmp.prev = e;
}

ListEntry.ListEntry(Node n) {
    node = n;
    next = null;
    prev = null;
}
```

Figure 4-4: Inserting an element in a list.

made to the `prev` field, however, is conditional. Based on whether `this.head` is `null` or not (that is, whether the list is empty or not), the `prev` field of `this.head` may be changed to `New$ListEntry`. Since the mapping `(this.head -> New$ListEntry)` may or may not be included in the final `prev'` relation, it is added to the upper bound of `prev'`. Because the final mapping of `this.head` under `prev'` is unknown, all possible mappings of `this.head` are removed from the lower bound of `prev'`.

Figures 4-5 to 4-7 revisit the methods called by the topological sort example of Chapter 1. The `init` method (Figure 4-5) initializes the `visitedInsNum` field of all the nodes of the receiver graph to the value zero. It involves a simple loop that matches the loop pattern optimized by our analysis: a linked data structure is traversed to the end to update some fields of its elements to a constant value. The pattern is identified by the loop statement `c = c.next` that makes the loop variable `c` visit all the elements of the list, and the loop condition `(c != null)` that checks whether the loop variable has got to the end of the list. For loops matching the pattern, the set of elements visited by the loop variable can be obtained precisely. In this example,

```
/*
 visitedInsNum' = visitedInsNum ++ (this.nodes.head.*next.node --> 0)
*/
void Graph.init() {
    ListEntry c = this.nodes.head;
    while (c != null) {
        c.node.visitedInsNum = 0;
        c = c.next;
    }
}
```

Figure 4-5: Initializing.

```
/*
 visitedInsNum' ⊆ visitedInsNum + (n.outNodesList.head.*next.node --> Int)
 visitedInsNum' ⊇ visitedInsNum - (n.outNodesList.head.*next.node --> Int)
*/
void Graph.fixIns(Node n) {
    ListEntry p = n.outNodesList.head;
    while (p != null) {
        p.node.visitedInsNum = p.node.visitedInsNum + 1;
        p = p.next;
    }
}
```

Figure 4-6: Integer arithmetic.

the set of list entries visited by `c` is given by the expression

`(this.nodes.head.*next & (ListEntry - null))`

that is, all non-null entries reachable from the head of `this.nodes`. Thus, the expression

`(this.nodes.head.*next & (ListEntry - null)).node`

denotes the exact set of nodes whose `visitedInsNum` fields are updated. As explained in Section 4.4.3, this expression is further simplified to the equivalent expression `this.nodes.head.*next.node`. Since the updated field (`visitedInsNum`) gets a constant value, namely zero, the expression generated for `visitedInsNum'` is precise, and thus the generated summary is a full specification of the method's behavior.

The `fixIns` method, given in Figure 4-6, increments the `visitedInsNum` field of all the nodes in the `outNodesList` of the node argument `n` by one. Similar to the previous case, this loop traverses a linked data structure to the end. Thus, the

```
/*
 findSource$return ⊆ entry.*next & (null + node.inDeg.(entry.*next.node.visitedInsNum))
 findSource$return ⊇ none
*/
ListEntry Graph.findSource(ListEntry entry) {
    ListEntry e = entry;
    while ((e != null) && (e.node.inDeg != e.node.visitedInsNum))
        e = e.next;
    return e;
}
```

Figure 4-7: Searching a list.

expression n.outNodesList.head.*next.node gives the exact set of nodes whose visitedInsNum fields are updated. However, because the updating value is not constant with respect to the loop, the specification inferred for visitedInsNum' is not precise. In fact, because the update involves integer arithmetic, our technique cannot infer much about the final value. Thus, the inferred summary only specifies that the visitedInsNum field of all nodes in the n.outNodesList.head.*next.node set are updated to some unknown integer. Because the final values of these fields are unknown, the lower bound does not contain any mappings for them.

The findSource method (Figure 4-7) illustrates how our analysis exploits loop conditions to produce more precise summaries. It searches the entries reachable from a given list entry (entry) to find a node whose inDeg and visitedInsNum fields are equal, and returns the entry that stores that node, or null if no such node exists. The analysis infers that, in each iteration, the variable e points to some object reachable from entry by following the next link, and thus e's final value (the return value of the method) must be in entry.*next. Furthermore, e's final value must violate the loop condition. That is, it must either be null or its node's inDeg and visitedInsNum fields must be equal. The analysis is not precise enough to infer the latter condition. However, it can infer that e's final value has a node whose inDeg field equals the visitedInsNum field of some node reachable from entry. The set of all such list entries is given by node.inDeg.(entry.*next.node.visitedInsNum). Thus, all objects violating the loop condition are included in the set encoded by

```
(null + node.inDeg.(entry.*next.node.visitedInsNum))
```

Therefore, the returned list entry belongs to the intersection of `entry.*next` and the above set.

4.4 Abstraction Technique

We use the abstract interpretation framework [13] to generate procedure summaries. In this section, we describe our abstraction technique by first defining the concrete and abstract domains, and then giving the transfer function that abstracts different program statements.

4.4.1 Definitions

Environments. The set of concrete values used in an object-oriented program comprises the values of variables, fields, and types. The concrete value of a variable is an object (which is denoted by a singleton set to provide a uniform definition); the concrete value of a field is a mapping from objects to objects; and the concrete value of a type is the set of all objects of that type. If *Obj* represents the set of all objects in a program, the set of all possible concrete values *CVal* is defined as follows[1]:

$$\mathit{CVal} = 2^{\mathit{Obj}} \cup 2^{(\mathit{Obj} \times \mathit{Obj})}$$

where 2^S denotes the powerset of a set S.

The set of abstract values, *AVal*, comprises the set of relational expressions defined by the grammar of Figure 4-1:

$$\mathit{AVal} = \mathit{Expr}$$

A concrete state maps each variable, field, and type to a concrete value. We use C to denote the set of all possible well-typed concrete states of a program:

$$C = \mathit{Var} \cup \mathit{Field} \cup \mathit{Type} \rightarrow \mathit{CVal}$$

[1] Arrays and maps require ternary relations that are not discussed in this chapter.

The abstract domain A consists of pairs of environments. That is, the set of all concrete states that may occur at a program point is abstracted with a pair of environments $\langle E^l,\, E^u \rangle$ where E^l and E^u represent the lower and upper bounds of relations, respectively. An environment is a mapping from each variable, field, and type to an abstract value:

$$\begin{aligned} A &= \mathit{Env} \times \mathit{Env} \\ \mathit{Env} &= \mathit{Var} \cup \mathit{Field} \cup \mathit{Type} \rightarrow \mathit{AVal} \end{aligned}$$

The set of concrete states at a program point is abstracted using the independent attribute method [32]: an abstract environment maps each variable, field, and type to an abstraction of the set of its values in the given concrete states. We therefore lose the correlation between the values of different variables at a program point.

The meaning of the abstraction is given by the concretization function γ defined as follows:

$$\begin{aligned} &\gamma : A \rightarrow 2^C \\ &\gamma(\langle E^l,\, E^u \rangle) = \{c \mid \exists c_0, \delta \mid \forall x.\ [\![E^l(x)]\!]^{\delta}_{c_0} \subseteq c(x) \subseteq [\![E^u(x)]\!]^{\delta}_{c_0}\} \end{aligned}$$

where c_0 is a well-typed initial state, δ is a binding of allocated symbolic objects to concrete objects not used in c_0, and $[\![e]\!]^{\delta}_{c_0}$ denotes the meaning of a relational expression e under the bindings defined by c_0 and δ. The concretization function γ gives the set of all concrete states in which the values of all variables, fields, and types are bounded from below by E^l and from above by E^u.

Lattice of Environments. We define a partial order $\sqsubseteq$ over pairs of environments as follows:

$$\langle E^l_1,\, E^u_1 \rangle \sqsubseteq \langle E^l_2,\, E^u_2 \rangle \iff \forall x.[\![E^l_1(x) \boxed{\supseteq} E^l_2(x)]\!] \wedge [\![E^u_1(x) \boxed{\subseteq} E^u_2(x)]\!]$$

where the expression $op_1 \boxed{\circ}\, op_2$ gives a syntactic concatenation of the operator $\circ$ with the operands op_1 and op_2, and $[\![f]\!]$ means that the relational formula f is a tautology in the theory of relations.

This partial order defines a pointed lattice over the set of all pairs of environments,

```
$\langle E^l_1,\ E^u_1\rangle \bigtriangledown \langle E^l_2,\ E^u_2\rangle =$ let $\langle E^l_3,\ E^u_3\rangle = \langle E^l_1,\ E^u_1\rangle \sqcup \langle E^l_2,\ E^u_2\rangle$ in
$\langle\ \lambda v : \mathit{Var}.$ if $\exists x, c, r \mid E^l_3(v) = x \boxed{\&} (x \boxed{\&} c).r$ then $\emptyset$            intersection
        elseif $\exists x, r \mid E^l_3(v) = x \boxed{\&} x.r \boxed{\&} .. \boxed{\&} x.r^{(k)}$ then $\emptyset$       multiple joins
        elseif $|E^l_3(v)| \geq n$ then $\emptyset$                                   length
        else $E^l_3(v)$
$\cup\ \lambda f : \mathit{Field}.$ if $|E^l_3(f)| \geq n$ then $\emptyset$                               length
        else $E^l_3(f)$
$\cup\ \lambda t : \mathit{Type}.\ E^l_3(t)$,

   $\lambda v : \mathit{Var}.$ if $\exists x, c, r \mid E^u_3(v) = x \boxed{+} (x \boxed{\&} c).r$ then $x \boxed{+} x.r$      intersection
        elseif $\exists x, r \mid E^u_3(v) = x \boxed{+} x.r \boxed{+} .. \boxed{+} x.r^{(k)}$ then $x.*r$  multiple joins
        elseif $|E^u_3(v)| \geq n$ then $\mathit{type}(v)$                             length
        else $E^u_3(v)$
$\cup\ \lambda f : \mathit{Field}.$ if $|E^u_3(f)| \geq n$ then $\mathit{domainType}(f) \boxed{\text{->}} \mathit{rangeType}(f)$       length
        else $E^u_3(f)$
$\cup\ \lambda t : \mathit{Type}.$ if $\exists l_1, .., l_m : \mathit{SymObj} \mid (E^u_3(t) = l_1 \boxed{+} .. \boxed{+} l_m)$
        then $E^u_3(t) \boxed{+} \mathit{symObjSet}(t)$                               allocation
        else $E^u_3(t)\ \rangle$
```

Figure 4-8: Widening rules.

with a top $\top = \langle E_\emptyset,\ E_{univ}\rangle$ and a bottom $\bot = \langle E_{univ},\ E_\emptyset\rangle$ where $E_\emptyset$ represents the environment in which everything is mapped to the empty expression, and E_{univ} denotes the environment in which everything is mapped to a relation that represents the universe of all atoms.

The lattice join $\sqcup$ is defined as follows:

$$\langle E^l_1,\ E^u_1\rangle \sqcup \langle E^l_2,\ E^u_2\rangle = \langle\ \lambda x.\ E^l_1(x) \boxed{\&} E^l_2(x),\ \ \lambda x.\ E^u_1(x) \boxed{+} E^u_2(x)\ \rangle$$

It computes the least upper bound of two pairs of environments by forming relational expressions that intersect the lower bounds of all variables, fields, and types in the two environments, and union their upper bounds.

We also define a widening operation – given in Figure 4-8 – to stabilize infinitely ascending chains in the lattice. Two pairs of environments are widened by first computing their least upper bound, and then approximating the result using the following four rules:

- **intersection.** For a homogeneous binary relation r and two expressions x and

c, a lower bound expression $x \boxed{\&} (x \boxed{\&} c).r$ is simplified to the empty set, and an upper bound expression $x \boxed{+} (x \boxed{\&} c).r$ is widened to the expression $x \boxed{+} x.r$. This widening becomes handy in summarizing loop bodies: let x denote the expression corresponding to a loop variable v at the beginning of the loop. If v is a pointer traversing a linked structure, its value after one iteration of the loop will be $(x \boxed{\&} c).r$ where c represents the loop condition, and r represents the link being traversed. Thus, the value of v after computing the least upper bound will be given by the lower bound $x \boxed{\&} (x \boxed{\&} c).r$ and the upper bound $x \boxed{+} (x \boxed{\&} c).r$. Dropping the intersecting condition c from the upper bound makes the expression amenable to the widening rule explained below which infers a transitive closure for v.

- **multiple joins.** An intersection of k joins $x \boxed{\&} x.r \boxed{\&} .. \boxed{\&} x.r^{(k)}$ in the lower bound environment is simplified to the empty set. A union of k joins $x \boxed{+} x.r \boxed{+} .. \boxed{+} x.r^{(k)}$ in the upper bound environment is widened to $x. * r$, where $*r$ is the reflexive transitive closure of r. In order to apply this rule, r need not be a single field name; it can be any arbitrary relational expression.

- **allocation.** If m objects are already allocated for a type t, i.e. $(E^u(t) = l_1 \boxed{+} .. \boxed{+} l_m)$ for some symbolic objects $l_1, .., l_m$, the upper bound of t is widened to include the symbolic object set of that type (denoted by *symObjSet*(t)) to be used in future allocations.

- **length.** Any lower bound expression whose number of operators is greater than a limit n is simplified to the empty set, and any such upper bound is widened to a universal relation of the appropriate type. Figure 4-8 uses the expression *type*(v) to denote the relation corresponding to the type of a variable v. Similarly, for a field f, this figure uses the expressions *domainType*(f) and *rangeType*(f) to denote the relations corresponding to the types of the declaring class and the target class of f, respectively.

Initial Abstraction. Summarizing a procedure p starts by generating a pair of initial abstract environments $\langle E^l_0, E^u_0 \rangle$ in which $E^l_0 = E^u_0$ and each formal parameter, field, and type accessed in p is mapped to a fresh relation represented by a symbolic name. Our analysis makes no assumptions about the possible aliasings between names, and thus the results are valid for all possible aliasings.

4.4.2 Transfer Function

The transfer function

$$\bar{\mathcal{F}} : \mathit{Stmt} \longrightarrow (\mathit{Env} \times \mathit{Env}) \longrightarrow (\mathit{Env} \times \mathit{Env})$$

models the effects of the program statements conforming to the grammar of Figure 3-1 on the abstract environments. The expressions generated by this function are syntactically simplified using the equivalence-preserving transformations described in the next section (4.4.3).

Evaluation of expressions: We use the auxiliary functions eval^l and eval^u – given in Figure 4-9(a) – to evaluate a program expression in a lower bound and an upper bound environment respectively. These functions differ from each other only in the way they treat integer arithmetic. This is because the programming language that we analyze (see the grammar of Figure 3-1) allows only simple expressions, and not compound ones such as conditional expressions. Therefore, the evaluation of a non-arithmetic expression in an environment is always exact. The function eval^l defines the lower bound of any integer arithmetic expression to be `none`, the empty set, while the function eval^u defines its upper bound to be `Int`, the set of all integer numbers.

Non-arithmetic program expressions are evaluated using the auxiliary function *eval* given in Figure 4-9(b). The evaluation of a constant is always a unary relation corresponding to that constant. The evaluation of a program variable is just a lookup in the environment, and the evaluation of a navigation expression is defined recursively by looking up the fields in the environment.

Sequence of statements: Two consecutive statements are abstracted by first abstracting the first statement, and then abstracting the second one using the result-

$$
\begin{array}{l}
\mathit{eval}^l : \mathit{PExpr} \rightarrow \mathit{Env} \rightarrow \mathit{Expr} \\
\mathit{eval}^l(e_1 + e_2, E) = \mathit{none} \\
\mathit{eval}^l(e_1 - e_2, E) = \mathit{none} \\
\mathit{eval}^l(e, E) = \mathit{eval}(e, E) \\
\\
\mathit{eval}^u : \mathit{PExpr} \rightarrow \mathit{Env} \rightarrow \mathit{Expr} \\
\mathit{eval}^u(e_1 + e_2, E) = \mathit{Int} \\
\mathit{eval}^u(e_1 - e_2, E) = \mathit{Int} \\
\mathit{eval}^u(e, E) = \mathit{eval}(e, E)
\end{array}
$$

(a)

$$
\begin{array}{l}
\mathit{eval} : \mathit{PExpr} \rightarrow \mathit{Env} \rightarrow \mathit{Expr} \\
\mathit{eval}(c : \mathit{PConst}, E) = \{\texttt{<c>}\} \\
\mathit{eval}(i : \mathit{IntConst}, E) = \mathit{Int}(i) \\
\mathit{eval}(v : \mathit{Var}, E) = E(v) \\
\mathit{eval}(e.f, E) = \mathit{eval}(e, E).E(f)
\end{array}
$$

(b)

Figure 4-9: Auxiliary functions to evaluate program expressions in an environment.

ing environments. That is,

$$\bar{\mathcal{F}}(S_1;\ S_2, \langle E^l,\ E^u \rangle) = \bar{\mathcal{F}}(S_2, \bar{\mathcal{F}}(S_1, \langle E^l,\ E^u \rangle))$$

Assignments to locals: Assigning an expression e to a local variable v is abstracted by the following rule:

$$\bar{\mathcal{F}}(v\ \texttt{=}\ e, \langle E^l,\ E^u \rangle) = \ \langle E^l[v \mapsto \mathit{eval}^l(e, E^l)],\ E^u[v \mapsto \mathit{eval}^u(e, E^u)] \rangle$$

That is, the lower and upper bounds of e become the lower and upper bounds of v.

Field updates: Assigning an expression e_2 to a field f of the object described by an expression e_1 is abstracted by the following rule:

$$
\begin{array}{l}
\bar{\mathcal{F}}(e_1.f\ \texttt{=}\ e_2, \langle E^l,\ E^u \rangle) = \\
\textbf{let } x_1^l = \mathit{eval}^l(e_1, E^l), x_1^u = \mathit{eval}^u(e_1, E^u),\ x_2^l = \mathit{eval}^l(e_2, E^l), x_2^u = \mathit{eval}^u(e_2, E^u) \textbf{ in} \\
\quad \textbf{if } (x_1^l = x_1^u) \wedge (x_2^l = x_2^u) \\
\quad \textbf{then } \langle E^l[f \mapsto E^l(f) \boxed{\texttt{++}} (x_1^l \boxed{\texttt{->}} x_2^l)],\ E^u[f \mapsto E^u(f) \boxed{\texttt{++}} (x_1^u \boxed{\texttt{->}} x_2^u)] \rangle \\
\quad \textbf{else } \langle E^l[f \mapsto E^l(f) \boxed{\texttt{-}} (x_1^u \boxed{\texttt{->}} \mathit{rangeType}(f))],\ E^u[f \mapsto E^u(f) \boxed{\texttt{+}} (x_1^u \boxed{\texttt{->}} x_2^u)] \rangle
\end{array}
$$

Two cases are distinguished:

- Strong update: if the lower bounds of e_1 and e_2 are syntactically the same as their upper bounds, then the computed values for both expressions are exact.

In this case, the value of e_2 overrides the previous value of $e_1.f$.

- Weak update: if the values of e_1 and e_2 are not necessarily exact, it is not clear for which object the f field was mutated, or how it was mutated. Thus, the upper bound allows the f field of any of the objects represented by the upper bound of e_1 to be mapped to any of the objects represented by the upper bound of e_2. The lower bound leaves all possible mutated objects unconstrained.

Allocations: We assume that an allocation statement `v = `**new** `T(`e_1`, .., `e_n`)` is broken into two consecutive statements[2]: `v = `**new** `T; v.init(`e_1`, .., `e_n`)` The first statement does the actual allocation, and the second one calls the constructor on the allocated object. Here, we explain the abstraction of the allocation statement. The call to the constructor will be abstracted similarly to other procedure calls (explained later).

$$\begin{aligned}
&\bar{\mathcal{F}}(v\ \texttt{=}\ \mathbf{new}\ t, \langle E^l,\ E^u\rangle) = \\
&\quad \mathbf{if}\ \mathit{containsSymObjSet}(t, E^u)\ \mathbf{then\ let}\ s = \mathit{symObjSet}(t)\ \mathbf{in} \\
&\qquad \langle E^l[v \mapsto \emptyset],\ E^u[v \mapsto s]\rangle \\
&\quad \mathbf{else\ let}\ s = \mathit{symObj}(t, E^u)\ \mathbf{in} \\
&\qquad \langle E^l[v \mapsto s, t \mapsto E^l(t) \boxed{+} s],\ E^u[v \mapsto s, t \mapsto E^u(t) \boxed{+} s]\rangle
\end{aligned}$$

The function *symObjSet*(t) represents the symbolic object set of type t, the function *containsSymObjSet*(t, E) returns true if that symbolic set is already used in the environment E (and false otherwise), and the function *symObj*(t, E) generates a fresh symbolic object of type t that is not already used in E.

Allocating an object of type t is abstracted by distinguishing two cases:

- If, as a result of a previous widening, a symbolic object set has already been generated for a type t, meaning that we have already hit the limit for enumerating new objects, the upper bound of the new object will be that same symbolic set and its lower bound will be the empty set. Since the cardinality of this symbolic set is unconstrained, no other updates are necessary.

[2] As done anyway, e.g. in the Soot framework [57].

- If no symbolic set has been generated, we generate a fresh symbol as the exact value of the new object. This symbol will then be added to the allocation expression of t.

Call sites: During the analysis, different calls to a procedure p may have different summaries. This is because the abstractions of field updates and allocation statements are based on whether the values computed for variables, fields, and types are exact or not. However, it is often the case that the summaries are structurally identical, and we can avoid re-computing them from scratch at each call site. One can predict whether or not two summaries will be structurally identical by looking at their *context*, an abstraction that records the parts of the environments with exact values:

$$\begin{aligned}
&\mathit{context}(\langle E^l,\ E^u\rangle) = \\
&\quad \{v : \mathit{Var} \mid E^l(v) = E^u(v)\} \cup \\
&\quad \{f : \mathit{Field} \mid E^l(f) = E^u(f)\} \cup \\
&\quad \{t : \mathit{Type} \mid \mathit{containsSymObjSet}(t, E^u) = \mathit{false}\}
\end{aligned}$$

Given a context of a call to p, our technique generates a *summary template* which can then be instantiated to produce a summary for a call to p.

Consider a call to a procedure p in an environment pair $\langle E^l,\ E^u\rangle$ with a context c. We compute a summary template t from c by first generating a pair of environments whose context is the same as c, but maps all variables, fields, and types to fresh symbolic values, and then summarizing the body of p on that generated pair of environments:

$$\begin{aligned}
&\mathit{template}(p, c) = \\
&\quad \textbf{let } E_1^l = \lambda x.\ \mathit{sym}(x), \\
&\qquad E_1^u = \ \lambda v.\ \textbf{if } (v \in c) \textbf{ then } E_1^l(v) \textbf{ else } \mathit{sym}(v) \\
&\qquad\qquad \cup\ \lambda f.\ \textbf{if } (f \in c) \textbf{ then } E_1^l(f) \textbf{ else } \mathit{sym}(f) \\
&\qquad\qquad \cup\ \lambda t.\ \textbf{if } (t \in c) \textbf{ then } E_1^l(t) \textbf{ else } E_1^l(t) \boxed{+} \mathit{symObjSet}(t) \\
&\quad \textbf{in } \bar{\mathcal{F}}(p\_\textsf{body}, \langle E_1^l,\ E_1^u\rangle)
\end{aligned}$$

where $\mathit{sym}(x)$ generates a fresh symbolic value for x.

The summary of the call site is then computed by instantiating the summary template t with the values from the environments $\langle E^l,\ E^u\rangle$ that represent the pre-state of the site. That is, by substituting the values of actual parameters and fields at the call site for the symbols used in the summary template.

$$\begin{aligned}&\bar{\mathcal{F}}(p, \langle E^l,\ E^u\rangle) = \mathit{instantiate}(\\ &\quad \mathit{template}(p, \mathit{context}(\langle E^l,\ E^u\rangle)), \langle E^l,\ E^u\rangle)\end{aligned}$$

The summary template t is saved and associated with the context c. If another call to p is later encountered with the same context c, then its summary can be computed by instantiating t with the values from its environment, rather than computing the summary directly from scratch.

If a procedure p has i formal parameters, accesses j different fields, and allocates k different types, then it has 2^{i+j+k} possible contexts. It should be noted that the empty context represents the case where none of the computed values are exact. Therefore, the template summary generated for the empty context only performs weak updates and although it is not the best approximation, it can be used in all other contexts too. To balance between precision and performance, we summarize each procedure on demand for its first l distinct contexts where l is a constant parameter of the analysis. Any further calls whose corresponding contexts are not visited before will be replaced by the procedure's template summary for the empty context.

Return statements: A procedure named `proc` is allocated a special variable `proc$return` to hold its return value. A return statement is simply an assignment of a value to that variable.

$$\begin{aligned}&\bar{\mathcal{F}}(\textbf{return}\ e, \langle E^l,\ E^u\rangle) =\\ &\quad\langle\ E^l[\texttt{proc\$return} \mapsto\ (E^l(\texttt{proc\$return})\ \boxed{\&}\ \mathit{eval}(e, E^l))],\\ &\quad\ \ E^u[\texttt{proc\$return} \mapsto\ (E^u(\texttt{proc\$return})\ \boxed{+}\ \mathit{eval}(e, E^u))]\ \rangle\end{aligned}$$

If the procedure has multiple return statements, the return values accumulate in E^l and E^u by intersecting the lower bounds and unioning the upper bounds.

Branches: Conditional statements are abstracted by first abstracting each branch independently, then combining the results using the lattice least upper bound operator.

$$\bar{\mathcal{F}}(\textbf{if } (c)\ S_1 \textbf{ else } S_2, \langle E^l,\ E^u\rangle) = \bar{\mathcal{F}}(S_1, \langle E^l,\ E^u\rangle) \sqcup \bar{\mathcal{F}}(S_2, \langle E^l,\ E^u\rangle)$$

Loops: Loops are abstracted by successively abstracting the body and joining each new abstraction with the abstraction of previous iterations. To produce more precise summaries, we intersect the loop condition with the variables used in it at the beginning of each iteration. Furthermore, the termination condition (negation of loop condition) is intersected with the final value of the variables in the loop condition.

$$\begin{array}{l}
\bar{\mathcal{F}}(\textbf{while}(c)\ S,\ \langle E^l,\ E^u\rangle) = \\
\quad \textbf{let } \langle E^l_1,\ E^u_1\rangle = \textbf{fix } \langle E^l,\ E^u\rangle, \\
\quad\quad \bar{\mathcal{F}}(S, \mathit{addCond}(c, \langle E^l,\ E^u\rangle)) \bigtriangledown \langle E^l,\ E^u\rangle \\
\quad \textbf{in } \mathit{addCond}(\neg c, \langle E^l_1,\ E^u_1\rangle)
\end{array}$$

where $\bigtriangledown$ is the lattice widening operator, and *addCond*$(c, \langle E^l,\ E^u\rangle)$ is a function that intersects a relational encoding of a condition c with the values of c's variables in the environments E^l and E^u. The fixpoint operator iteratively abstracts the loop body and joins the resulting pair of environments with the previous pairs until the expressions stored in the environments are unchanged, i.e a fixpoint is reached.

Figure 4-10 gives the definition of the *addCond* function. Given a loop condition c and a pair of environments $\langle E^l,\ E^u\rangle$, whenever possible, it updates the relational expressions of c's variables to the set of tuples for which c evaluates to true. Since we always abstract arithmetic expressions by the set of all integers (`Int`), the *addCond* function cannot exploit integer comparison tests ($e_1 < e_2$). All other loop conditions, however, are exploited to make the environments more precise. A simple non-arithmetic loop condition can be written as ($v.f_1..f_n$ *op* e) where v is a variable, f_1 to f_n are fields, e is a program expression, and *op* is either an equality test (== and !=) or a test on the type of the object (`instanceof`). Based on the operator

```
$addCond(e_1 < e_2, \langle E^l,\ E^u \rangle) = \langle E^l,\ E^u \rangle$

$addCond(v.f_1 \ldots f_n$ == $e, \langle E^l,\ E^u \rangle) =$
 let $x_1 = E^l(v),\ y_1 = eval^l(e, E^l),\ x_2 = E^u(v),\ y_2 = eval^u(e, E^u)$
 in $\langle E^l[v \mapsto x_1 \boxed{\&} E^l(f_1) \boxdot \cdot\cdot \boxdot E^l(f_n) \boxdot y_1],\ E^u[v \mapsto x_2 \boxed{\&} E^u(f_1) \boxdot \cdot\cdot \boxdot E^u(f_n) \boxdot y_2] \rangle$

$addCond(v.f_1 \ldots f_n$ != $e, \langle E^l,\ E^u \rangle) =$
 let $x_1 = E^l(v),\ y_1 = eval^l(e, E^l),\ x_2 = E^u(v),\ y_2 = eval^u(e, E^u)$
 in if $y_1 = y_2$
    then $\langle E^l[v \mapsto x_1 \boxed{\&} E^l(f_1) \boxdot \cdot\cdot \boxdot E^l(f_n) \boxdot (type(e) \boxed{-} y_1)],$
         $E^u[v \mapsto x_2 \boxed{\&} E^u(f_1) \boxdot \cdot\cdot \boxdot E^u(f_n) \boxdot (type(e) \boxed{-} y_2)] \rangle$
    else $\langle E^l,\ E^u \rangle$

$addCond(v.f_1 \ldots f_n$ instanceof $T, E^l) =$
 let $x_1 = E^l(v), y_1 = E^l(T), x_2 = E^u(v), y_2 = E^u(T)$
 in $\langle E^l[v \mapsto x_1 \boxed{\&} E^l(f_1) \boxdot \cdot\cdot \boxdot E^l(f_n) \boxdot y_1],\ E^u[v \mapsto x_2 \boxed{\&} E^u(f_1) \boxdot \cdot\cdot \boxdot E^u(f_n) \boxdot y_2] \rangle$

$addCond(c_1$ && $c_2, \langle E^l,\ E^u \rangle) = addCond(c_1, \langle E^l,\ E^u \rangle) \boxed{\&} addCond(c_2, \langle E^l,\ E^u \rangle)$

$addCond(c_1$ || $c_2, \langle E^l,\ E^u \rangle) = addCond(c_1, \langle E^l,\ E^u \rangle) \boxed{+} addCond(c_2, \langle E^l,\ E^u \rangle)$

$\langle E_1^l,\ E_1^u \rangle \boxed{+} \langle E_2^l,\ E_2^u \rangle = \langle \lambda x.\ E_1^l(x) \boxed{+} E_2^l(x),\ \lambda x.\ E_1^u(x) \boxed{+} E_2^u(x) \rangle$
$\langle E_1^l,\ E_1^u \rangle \boxed{\&} \langle E_2^l,\ E_2^u \rangle = \langle \lambda x.\ E_1^l(x) \boxed{\&} E_2^l(x),\ \lambda x.\ E_1^u(x) \boxed{\&} E_2^u(x) \rangle$
```

Figure 4-10: The auxiliary function to add loop conditions to loop variables.

op, the *addCond* function computes a relational expression that represents the set of tuples that pass the given condition. This set is then intersected with the relational expressions of v in the given environments to represent only those values of v for which the loop condition evaluates to true.

We provide an example to clarify how *addCond* filters the values of loop variables using the loop condition. Consider a loop accessing variables c and d, and a field f whose values at the beginning of the loop are approximated by expressions c_0, d_0, and f_0 respectively. If the loop condition is $(c.f{=}{=}d)$, then the subset of c_0 that passes the condition in the first iteration of the loop belongs to $f_0.d_0$, the set of atoms whose mapping under f_0 is an atom in d_0. The set of values that pass the loop condition is therefore $(c_0 \boxed{\&}\, f_0.d_0)$. The pair of environments obtained by applying the *addCond* function maps c to this more accurate expression computed with respect to both E^l and E^u.

Now suppose that the loop condition is $(c.f\,!{=}d)$. The subset of c_0 that passes this condition is the set of atoms whose mapping under f_0 is not equal to d_0. If d_0 is the exact value of d (that is, $E^l(d) = E^u(d) = d_0$), all values not equal to d_0 are given by $\mathit{type}(d) \boxed{-}\, d_0$. Therefore, the set of values that passes the loop condition can be encoded as $(c_0 \boxed{\&}\, f_0.(\mathit{type}(d) \boxed{-}\, d_0))$. However, if d_0 is not exact, that is, it encodes a set containing more than one value for d, then we cannot specify which values are not equal to d. Therefore, we do not filter the environment any further in this case.

4.4.3 Simplifications

Expressions generated by the transfer function can often be simplified using relational logic equivalence rules, which reduce the size of an expression without changing its semantics. Those rules are given in Figure 4-11.

Figure 4-11(a) gives a set of equivalence rules in relational logic that simplifies union, intersection, difference, Cartesian product, and composition of relations. The rules in Figure 4-11(b) simplify relational expressions based on the semantics of reflexive transitive closure and relational override.

Simplification rules of Figure 4-11(c) are based on the semantics of types and

$$\begin{array}{l}
x \boxed{+} x \longrightarrow x \\
x \boxed{-} (x \boxed{-} y) \longrightarrow x \boxed{\&} y \\
x.(x \boxed{\texttt{->}} y) \longrightarrow y \\
x.r \boxed{+} x.s \longrightarrow x.(r \boxed{+} s) \\
x.r \boxed{+} y.r \longrightarrow (x \boxed{+} y).r \\
(x \boxed{\&} c) \boxed{+} (y \boxed{\&} c) \longrightarrow (x \boxed{+} y) \boxed{\&} c \\
(x \boxed{\texttt{->}} z) \boxed{+} (y \boxed{\texttt{->}} z) \longrightarrow (x \boxed{+} y) \boxed{\texttt{->}} z
\end{array}$$

(a)

$$\begin{array}{l}
x.*r.*r \longrightarrow x.*r \\
x \boxed{+} (x.*r) \longrightarrow x.*r \\
(x \boxed{+} x.r).*r \longrightarrow x.*r \\
r \boxed{++} (x \boxed{\texttt{->}} y) \boxed{++} (x \boxed{\texttt{->}} z) \longrightarrow r \boxed{++} (x \boxed{\texttt{->}} z) \\
r \boxed{+} (r \boxed{++} (x \boxed{\texttt{->}} y)) \longrightarrow r \boxed{+} (x \boxed{\texttt{->}} y) \\
r \boxed{\&} (r \boxed{++} (x \boxed{\texttt{->}} y)) \longrightarrow r \boxed{-} (x \boxed{\texttt{->}} \mathit{rangeType}(r))
\end{array}$$

(b)

$$\begin{array}{l}
(x \boxed{\&} (\mathit{type}(x) \boxed{-} \mathit{null})).r \longrightarrow x.r \\
\mathit{type}(x).(x \boxed{\texttt{->}} y) \longrightarrow y \\
x.(\mathit{type}(x) \boxed{\texttt{->}} y) \longrightarrow y \\
x \boxed{\&} \mathit{type}(x) \longrightarrow x \\
r_{pre} \boxed{-} (x_{new} \boxed{\texttt{->}} y) \longrightarrow r_{pre} \\
x_{pre}.(r_{pre} \boxed{++} (y_{pre} \boxed{\texttt{->}} z) \boxed{++} (t_{new} \boxed{\texttt{->}} w)) \longrightarrow x_{pre}.(r_{pre} \boxed{++} (y_{pre} \boxed{\texttt{->}} z)) \\
x_{new}.(r_{pre} \boxed{++} (y_{pre} \boxed{\texttt{->}} z) \boxed{++} (x_{new} \boxed{\texttt{->}} w)) \longrightarrow w
\end{array}$$

(c)

Figure 4-11: The simplification rules used. The function *type*(e) denotes the relation corresponding to the type of an expression e. The expressions x_{new} and x_{pre} represent the cases where x is a newly allocated symbolic object and where it belongs to the pre-state, respectively. A name with no subscript may be either.

$$\begin{tikzcd}
\langle E^l_0, E^u_0\rangle \arrow[r, "\bar{\mathcal{F}}"] \arrow[d, "\gamma"] & \langle E^l_1, E^u_1\rangle \cdots \arrow[r, "\bar{\mathcal{F}}"] \arrow[d, "\gamma"] & \langle E^l_i, E^u_i\rangle \cdots \arrow[d, "\gamma"] \\
c_0 \arrow[r, "\mathcal{F}"] & c_1 \cdots \arrow[r, "\mathcal{F}"] & c_i \cdots
\end{tikzcd}$$

Figure 4-12: Relation between concrete states and abstract states

allocations. Their validity is based on the following facts: (1) a null object cannot be dereferenced, that is, *null.r* $= \emptyset$ for every relation r, and (2) newly allocated objects do not alias objects in the pre-state. Figure 4-11(c) uses x_{new} and x_{pre} to distinguish the case that x is an allocated symbolic object from the case where it is a relation in the pre-state. A name with no subscript may be either.

4.5 Properties

4.5.1 Safety

The abstraction described above is *safe*, meaning that the generated lower and upper bounds account for all executions of the summarized procedure.

Figure 4-12 shows the relation between the concrete states c_i and abstract environments $\langle E^l_i, E^u_i\rangle$ for a safe abstraction. In this figure, γ is the concretization function defined in Section 4.4.1 as:

$$\gamma : A \rightarrow 2^C$$
$$\gamma(\langle E^l, E^u\rangle) = \{c \mid \exists c_0, \delta \mid \forall x.\ [\![E^l(x)]\!]^{\delta}_{c_0} \subseteq c(x) \subseteq [\![E^u(x)]\!]^{\delta}_{c_0}\}$$

$\mathcal{F}$ is the concrete state transition function which can be defined by the operational semantics of the program statements as given in Figure 4-13, and $\bar{\mathcal{F}}$ is the abstract state transition function (the transfer function defined in Section 4.4.2).

In order to prove the safety property, it is sufficient to show that:

1. The initial abstraction is safe. That is,
   $\forall c : C \mid c \in \gamma(\langle E^l_0, E^u_0\rangle)$

2. The composition property holds. That is,
   $\mathcal{F} \circ \gamma \subseteq \gamma \circ \bar{\mathcal{F}}$

The initial abstraction is safe. This property is valid because the relations used to encode the pre-state are typed, and the symbolic names used for their values are uninterpreted. That is, they can be instantiated by any well-typed concrete values. More specifically, because both E_0^l and E_0^u map each variable, field, and type to a fresh relation constant, and because they contain no symbolically allocated objects, we have: $\llbracket E_0^l(x) \rrbracket_c^\delta = c(x)$ and $\llbracket E_0^u(x) \rrbracket_c^\delta = c(x)$ for any well-typed concrete state c and any arbitrary binding of symbolic objects to concrete objects δ. Therefore,

$$\forall c : C, \delta \mid \forall x.\ \llbracket E_0^l(x) \rrbracket_c^\delta \subseteq c(x) \subseteq \llbracket E_0^u(x) \rrbracket_c^\delta$$

That is, $\forall c : C \mid c \in \gamma(\langle E_0^l,\ E_0^u \rangle)$.

The composition property holds. This property can be proved by structural induction over statements: we first prove that it holds for simple statements, and then use the results to prove that it holds for compound statements too. The proofs require the following lemmas:

Lemma 2. *The $eval^l$ and $eval^u$ functions are safe. That is, the expressions $eval^l(e, E^l)$ and $eval^u(e, E^u)$ correctly bound all values of an expression e that may occur in the concretization of $\langle E^l,\ E^u \rangle$.*

$$\begin{aligned} &\forall c \in \gamma(\langle E^l,\ E^u \rangle).\exists c_0, \delta \mid \\ &\qquad \forall e : \mathit{ProgExpr}.\llbracket eval^l(e, E^l) \rrbracket_{c_0}^\delta \subseteq eval^c(e, c) \subseteq \llbracket eval^u(e, E^u) \rrbracket_{c_0}^\delta \end{aligned}$$

Proof. Consider a concrete state $c \in \gamma(\langle E^l,\ E^u \rangle)$. By the definition of γ, we have $\forall x.\ \llbracket E^l(x) \rrbracket_i^{\delta^*} \subseteq c(x) \subseteq \llbracket E^u(x) \rrbracket_i^{\delta^*}$ for some initial state i and some allocation binding δ^*. Using the definitions of $eval^l$, $eval^u$, and $eval^c$, it is easy to see that i and δ^* satisfy the lemma for all program expressions. That is,

$$\forall e : \mathit{ProgExpr}.\llbracket eval^l(e, E^l) \rrbracket_i^{\delta^*} \subseteq eval^c(e, c) \subseteq \llbracket eval^u(e, E^u) \rrbracket_i^{\delta^*}$$

$$\mathcal{F}(S, C^*) = \bigcup_{c \in C^*}\{\mathcal{F}(S, c)\}$$

$$\frac{\mathit{eval}^c(e, c) = e^*}{\mathcal{F}(v = e, c) = c[v \mapsto e^*]}$$

$$\frac{\mathit{eval}^c(e_1, c) = e_1^*,\ \mathit{eval}^c(e_2, c) = e_2^*,\ c(f) = f^*}{\mathcal{F}(e_1.f = e_2, c) = c[f \mapsto f^*\ \texttt{++}\ (e_1^*\ \texttt{->}\ e_2^*)]}$$

$$\frac{c(t) = t^*,\ \mathit{obj} \notin c(t)}{\mathcal{F}(v = \textbf{new}\ t, c) = c[v \mapsto \{\mathit{obj}\}, t \mapsto (t^* \cup \{\mathit{obj}\})]}$$

$$\frac{c[\mathit{forml}_1 \mapsto \mathit{eval}^c(e_1, c), \ldots, \mathit{forml}_n \mapsto \mathit{eval}^c(e_n, c)] = \bar{i},\ \mathcal{F}(\mathit{proc\_body}, \bar{i}) = c_2^*}{\mathcal{F}(\mathit{proc}(e_1, \ldots, e_n), c) = c_2^*}$$

$$\frac{c[\mathit{forml}_1 \mapsto \mathit{eval}^c(e_1, c), \ldots, \mathit{forml}_n \mapsto \mathit{eval}^c(e_n, c)] = \bar{i},\ \mathcal{F}(\mathit{proc\_body}, \bar{i}) = c_2^*}{\mathcal{F}(v = \mathit{proc}(e_1, \ldots, e_n), c) = c_2^*[v \mapsto \mathit{eval}^c(\mathit{proc\$return}, c_2^*)]}$$

$$\frac{\mathit{eval}^c(e, c) = e^*}{\mathcal{F}(\textbf{return}\ e, c) = c[\mathit{proc\$return} \mapsto e^*]}$$

$$\frac{\mathit{eval}^c(\mathit{cond}, c) = \{\mathit{true}\},\ \mathcal{F}(S_1, c) = c^*}{\mathcal{F}(\textbf{if}\ (\mathit{cond})\ S_1\ \textbf{else}\ S_2, c) = c^*}$$

$$\frac{\mathit{eval}^c(\mathit{cond}, c) = \{\mathit{false}\},\ \mathcal{F}(S_2, c) = c^*}{\mathcal{F}(\textbf{if}\ (\mathit{cond})\ S_1\ \textbf{else}\ S_2, c) = c^*}$$

$$\frac{\mathit{eval}^c(\mathit{cond}, c) = \{\mathit{false}\}}{\mathcal{F}(\textbf{while}\ (\mathit{cond})\ S, c) = c}$$

$$\frac{\mathit{eval}^c(\mathit{cond}, c) = \{\mathit{true}\},\ \mathcal{F}(S, c) = \bar{i},\ \mathcal{F}(\textbf{while}\ (\mathit{cond})\ S, \bar{i}) = c_2^*}{\mathcal{F}(\textbf{while}\ (\mathit{cond})\ S, c) = c_2^*}$$

$$\frac{\mathcal{F}(S_1, c) = \bar{i},\ \mathcal{F}(S_2, \bar{i}) = c_2^*}{\mathcal{F}(S_1; S_2, c) = c_2^*}$$

(a)

$\mathit{eval}^c : \mathit{PExpr} \rightarrow C \rightarrow \mathit{CVal}$
$\mathit{eval}^c(\mathit{const} : \mathit{PConst}, c) = \{\mathit{const}\}$
$\mathit{eval}^c(i : \mathit{IntConst}, c) = \{i\}$
$\mathit{eval}^c(v : \mathit{Var}, c) = c(v)$
$\mathit{eval}^c(e.f, c) = \mathit{eval}^c(e, c).c(f)$
$\mathit{eval}^c(e_1 + e_2, c) = \mathit{eval}^c(e_1, c) + \mathit{eval}^c(e_2, c)$
$\mathit{eval}^c(e_1 - e_2, c) = \mathit{eval}^c(e_1, c) - \mathit{eval}^c(e_2, c)$

(b)

Figure 4-13: (a) The definition of $\mathcal{F}$: an operational semantics on a concrete state $c \in C$. The first rule lifts the semantics to a set of concrete states $C^* \subseteq C$. (b) The auxiliary function $\mathit{eval}^c(e, c)$ evaluates an expression e on a concrete state c.

Therefore, the lemma holds. □

Lemma 3. *The operator $\sqcup$ is safe. That is, the result of $\sqcup$ is weaker than either of its arguments:*

$$\gamma(\langle E_1^l,\ E_1^u\rangle) \cup \gamma(\langle E_2^l,\ E_2^u\rangle) \subseteq \gamma(\langle E_1^l,\ E_1^u\rangle \sqcup \langle E_2^l,\ E_2^u\rangle)$$

Proof. Consider an arbitrary concrete state $c \in \gamma(\langle E_1^l,\ E_1^u\rangle)$. By the definition of γ, for some initial state i and some allocation binding δ^*, we have

$$\forall x.\ \llbracket E_1^l(x)\rrbracket_i^{\delta^*} \subseteq c(x) \subseteq \llbracket E_1^u(x)\rrbracket_i^{\delta^*}$$

Let $\bar{i}$ be an extension of i that includes some arbitrary initial values for any relation in E_2^l or E_2^u that does not appear in E_1^l or E_1^u. Similarly, let $\bar{\delta}^*$ be an extension of δ^* that includes an arbitrary fresh binding for any symbolically allocated object that appears in E_2^l or E_2^u but not in E_1^l or E_1^u. Since the mappings previously defined by i and δ^* are still the same under $\bar{i}$ and $\bar{\delta}^*$, we have $\forall x.\ \llbracket E_1^l(x)\rrbracket_{\bar{i}}^{\bar{\delta}^*} \subseteq c(x) \subseteq \llbracket E_1^u(x)\rrbracket_{\bar{i}}^{\bar{\delta}^*}$. By the properties of union and intersection in set theory, we have

$$\forall x.\ \llbracket E_1^l(x)\rrbracket_{\bar{i}}^{\bar{\delta}^*} \cap \llbracket E_2^l(x)\rrbracket_{\bar{i}}^{\bar{\delta}^*} \subseteq c(x) \subseteq \llbracket E_1^u(x)\rrbracket_{\bar{i}}^{\bar{\delta}^*} \cup \llbracket E_2^u(x)\rrbracket_{\bar{i}}^{\bar{\delta}^*}$$

By the semantics of the relational intersection and union, we have

$$\forall x.\ \llbracket E_1^l(x)\ \boxed{\&}\ E_2^l(x)\rrbracket_{\bar{i}}^{\bar{\delta}^*} \subseteq c(x) \subseteq \llbracket E_1^u(x)\ \boxed{+}\ E_2^u(x)\rrbracket_{\bar{i}}^{\bar{\delta}^*}$$

Therefore, $c \in \gamma(\langle E_1^l,\ E_1^u\rangle \sqcup \langle E_2^l,\ E_2^u\rangle)$, which implies that

$$\gamma(\langle E_1^l,\ E_1^u\rangle) \subseteq \gamma(\langle E_1^l,\ E_1^u\rangle \sqcup \langle E_2^l,\ E_2^u\rangle)$$

Similarly, one can show that

$$\gamma(\langle E_2^l,\ E_2^u\rangle) \subseteq \gamma(\langle E_1^l,\ E_1^u\rangle \sqcup \langle E_2^l,\ E_2^u\rangle)$$

Therefore, the lemma holds. □

Lemma 4. *The operator $\bigtriangledown$ is safe. That is, it produces a result weaker than either of its arguments:*

$$\gamma(\langle E^l_1,\, E^u_1\rangle) \cup \gamma(\langle E^l_2,\, E^u_2\rangle) \subseteq \gamma(\langle E^l_1,\, E^u_1\rangle \bigtriangledown \langle E^l_2,\, E^u_2\rangle)$$

Proof. The widening operator first computes the least upper bound of its two arguments, and then applies a set of widening rules. Lemma 3 proves that after applying the least upper bound operator, the result is weaker than either of the arguments. Furthermore, by the semantics of the relational operators, it is easy to see that the result of each widening rule is weaker than either of its arguments. Therefore, the $\bigtriangledown$ operator is safe. □

Lemma 5. *The addCond function is safe. That is, addCond accounts for all concrete states in which a condition evaluates to true:*

$$\begin{aligned}&\forall c \in \gamma(\langle E^l,\, E^u\rangle).\forall \mathit{cond}.\\ &\quad \mathit{eval}^{\mathcal{C}}(\mathit{cond}, c) = \{\mathit{true}\} \implies c \in \gamma(\mathit{addCond}(\mathit{cond}, \langle E^l,\, E^u\rangle))\end{aligned}$$

Proof. We prove the lemma for the case where the condition *cond* is of the form $v.f_1 \ldots f_n == e$. Other cases can be proved similarly. Consider a concrete state $c \in \gamma(\langle E^l,\, E^u\rangle)$ in which the condition holds. By the definition of γ, for some initial state i and some allocation binding δ^*, we have:

$$\forall x.\ [\![E^l(x)]\!]^{\delta^*}_i \subseteq c(x) \subseteq [\![E^u(x)]\!]^{\delta^*}_i$$

Let $\mathit{addCond}(\mathit{cond}, \langle E^l,\, E^u\rangle) = \langle E^l_{\mathit{post}},\, E^u_{\mathit{post}}\rangle$. Since *cond* is $(v.f_1 \ldots f_n == e)$, the *addCond* function only modifies the mapping of the variable v; other mappings are unchanged. Therefore, in order to prove the lemma, it is sufficient to show that

$$[\![E^l_{\mathit{post}}(v)]\!]^{\delta^*}_i \subseteq c(v) \subseteq [\![E^u_{\mathit{post}}(v)]\!]^{\delta^*}_i$$

Since *cond* holds in the state c, we have $c(v).c(f_1)\dots c(f_n) == eval^c(e,c)$ which is equivalent to $c(v) \subseteq c(f_1)\dots c(f_n).eval^c(e,c)$ following a relational view of the heap. Therefore, we have

$$c(v) = c(v) \cap c(f_1)\dots c(f_n).eval^c(e,c)$$

Since $c \in \gamma(\langle E^l,\ E^u\rangle)$ and by Lemma 2, we have

$$\begin{aligned}\llbracket E^l(v)\ \boxed{\&}\ E^l(f_1)\dots E^l(f_n).eval^l(e,E^l)\rrbracket_i^{\delta^*} &\subseteq c(v) \cap c(f_1)\dots c(f_n).eval^c(e,c)\\ &\subseteq \llbracket E^u(v)\ \boxed{\&}\ E^u(f_1)\dots E^u(f_n).eval^u(e,E^u)\rrbracket_i^{\delta^*}\end{aligned}$$

and therefore, $\llbracket E^l_{post}(v)\rrbracket_i^{\delta^*} \subseteq c(v) \subseteq \llbracket E^u_{post}(v)\rrbracket_i^{\delta^*}$. □

Given the above lemmas, we can prove that for any statement, the composition property holds. That is, for each statement S, the following property is valid:

$$\forall c.\langle E^l,\ E^u\rangle \mid c \in \mathcal{F}(S,\gamma(\langle E^l,\ E^u\rangle)) \implies c \in \gamma(\bar{\mathcal{F}}(S,\langle E^l,\ E^u\rangle))$$

The proof for each kind of statement is as follows:

Assignments to locals. Let the statement S be $v = e$. Consider a concrete state $c \in \mathcal{F}(S,\gamma(\langle E^l,\ E^u\rangle))$. By the definition of $\mathcal{F}$ (Figure 4-13), there exists some concrete state $c^* \in \gamma(\langle E^l,\ E^u\rangle)$ where $c = c^*[v \mapsto eval^c(e,c^*)]$. Because $c^* \in \gamma(\langle E^l,\ E^u\rangle)$, for some initial state i and some allocation binding δ^*, we have $\forall x.\ \llbracket E^l(x)\rrbracket_i^{\delta^*} \subseteq c^*(x) \subseteq \llbracket E^u(x)\rrbracket_i^{\delta^*}$.

Let $\bar{\mathcal{F}}(v = e, \langle E^l,\ E^u\rangle) = \langle E^l_{post},\ E^u_{post}\rangle$. Since the assignment statement only modifies the mapping of v, we have

$$\forall x \neq v \mid (c(x) = c^*(x)) \wedge (E^l_{post}(x) = E^l(x)) \wedge (E^u_{post}(x) = E^u(x))$$

and for $x = v$, we have $(c(v) = eval^c(e,c^*))$, $(E^l_{post}(v) = eval^l(e,E^l))$, and $(E^u_{post}(v) = eval^u(e,E^u))$. By Lemma 2, we have

$$\llbracket eval^l(e,E^l)\rrbracket_i^{\delta^*} \subseteq eval^c(e,c^*) \subseteq \llbracket eval^u(e,E^u)\rrbracket_i^{\delta^*}$$

Therefore

$$\forall x.\ [\![E^l_{post}(x)]\!]^{\delta^*}_i \subseteq c(x) \subseteq [\![E^u_{post}(x)]\!]^{\delta^*}_i$$

That is $c \in \gamma(\langle E^l_{post},\ E^u_{post}\rangle)$, and the composition property holds.

Field updates. Let the statement S be $e_1.f = e_2$. Consider a concrete state $c \in \mathcal{F}(S, \gamma(\langle E^l,\ E^u\rangle))$. By the definition of $\mathcal{F}$ (Figure 4-13), there exists some concrete state $c^* \in \gamma(\langle E^l,\ E^u\rangle)$ where $c = c^*[f \mapsto c^*(f)\ \texttt{++}\ (eval^c(e_1, c^*)\ \texttt{->}\ eval^c(e_2, c^*))]$. Because $c^* \in \gamma(\langle E^l,\ E^u\rangle)$, for some initial state i and some allocation binding δ^*, we have: $\forall x.\ [\![E^l(x)]\!]^{\delta^*}_i \subseteq c^*(x) \subseteq [\![E^u(x)]\!]^{\delta^*}_i$.

Let $\bar{\mathcal{F}}(e_1.f = e_2, \langle E^l,\ E^u\rangle) = \langle E^l_{post},\ E^u_{post}\rangle$, $x^l_1 = eval^l(e_1, E^l)$, $x^u_1 = eval^u(e_1, E^u)$, $x^l_2 = eval^l(e_2, E^l)$, and $x^u_2 = eval^u(e_2, E^u)$. We distinguish the following two cases:

1. Strong update. In this case $(x^l_1 = x^u_1)$ and $(x^l_2 = x^u_2)$. That is, x^l_1 (and thus x^u_1) and x^l_2 (and thus x^u_2) give the exact values of e_1 and e_2 and thus are singleton sets. By the definition of $\bar{\mathcal{F}}$, we have
   $E^l_1(f) = E^l(f)\ \boxed{\texttt{++}}\ (x^l_1\ \boxed{\texttt{->}}\ x^l_2)$, and $E^u_1(f) = E^u(f)\ \boxed{\texttt{++}}\ (x^u_1\ \boxed{\texttt{->}}\ x^u_2)$
   By Lemma 2, we have
   $[\![x^l_1]\!]^{\delta^*}_i \subseteq eval^c(e_1, c^*) \subseteq [\![x^u_1]\!]^{\delta^*}_i$, and $[\![x^l_2]\!]^{\delta^*}_i \subseteq eval^c(e_2, c^*) \subseteq [\![x^u_2]\!]^{\delta^*}_i$
   and because x^l_1, x^u_1, x^l_2, and x^u_2 are singletons, we have
   $[\![E^l_{post}(f)]\!]^{\delta^*}_i \subseteq c^*(f)\ \texttt{++}\ (eval^c(e_1, c^*)\ \texttt{->}\ eval^c(e_2, c^*)) \subseteq [\![E^u_{post}(f)]\!]^{\delta^*}_i$
   Therefore,
   $[\![E^l_{post}(f)]\!]^{\delta^*}_i \subseteq c(f) \subseteq [\![E^u_{post}(f)]\!]^{\delta^*}_i$

2. Weak update. In this case, x^l_1, x^u_1, x^l_2, and x^u_2 are not necessarily singletons. By the definition of $\bar{\mathcal{F}}$, we have $E^l_{post}(f) = E^l(f)\ \boxed{-}\ (x^u_1\ \boxed{\texttt{->}}\ rangeType(f))$,
   and $E^u_{post}(f) = E^u(f)\ \boxed{+}\ (x^u_1\ \boxed{\texttt{->}}\ x^u_2)$
   Again, by Lemma 2 and by the semantics of the relational operators, we have
   $[\![E^l_{post}(f)]\!]^{\delta^*}_i \subseteq c(f) \subseteq [\![E^u_{post}(f)]\!]^{\delta^*}_i$

Using the above results, and because

$$\forall x \neq f \mid (c(x) = c^*(x)) \wedge (E^l_{post}(x) = E^l(x)) \wedge (E^u_{post}(x) = E^u(x))$$

it is true that

$$\forall x.\ [\![E^l_{post}(x)]\!]_i^{\delta^*} \subseteq c(x) \subseteq [\![E^u_{post}(x)]\!]_i^{\delta^*}$$

That is, $c \in \gamma(\langle E^l_{post},\ E^u_{post}\rangle)$, and the composition property holds.

Allocations. Let the statement S be $v = \textbf{new}\ t$. Consider a concrete state $c \in \mathcal{F}(S, \gamma(\langle E^l,\ E^u\rangle))$. By the definition of $\mathcal{F}$ (Figure 4-13), there exists some concrete state $c^* \in \gamma(\langle E^l,\ E^u\rangle)$ where $c = c^*[v \mapsto \{obj\}, t \mapsto (c^*(t) \cup \{obj\})]$ for some fresh object *obj*. Because $c^* \in \gamma(\langle E^l,\ E^u\rangle)$, for some initial state i and some allocation binding δ^*, we have $\forall x.\ [\![E^l(x)]\!]_i^{\delta^*} \subseteq c^*(x) \subseteq [\![E^u(x)]\!]_i^{\delta^*}$

Let $\bar{\mathcal{F}}(v = \textbf{new}\ t, \langle E^l,\ E^u\rangle) = \langle E^l_{post},\ E^v_{post}\rangle$. If a symbolic object set is already used in the abstraction of t, i.e. $\mathit{containsSymObjSet}(t, E^u) = \mathit{true}$, we define s to be that symbolic set (given by $\mathit{symObjSet}(t)$), and $\bar{\delta^*} = \delta^*[s \mapsto (\delta^*(s) \cup \{obj\})]$.
By the definition of $\bar{\mathcal{F}}$, in this case, we have $(E^l_{post}(v) = \emptyset)$ and $(E^u_{post}(v) = s)$. Therefore,

$$[\![E^l_{post}(v)]\!]_i^{\bar{\delta^*}} \subseteq c(v) \subseteq [\![E^u_{post}(v)]\!]_i^{\bar{\delta^*}}$$

If the symbolic set is not used already, i.e. $\mathit{containsSymObjSet}(t, E^u) = \mathit{false}$, by the definition of $\bar{\mathcal{F}}$, $(E^l_{post}(v) = s)$, $(E^u_{post}(v) = s)$, $(E^l_{post}(t) = E^l(t) \boxed{+} s)$, and $(E^u_{post}(t) = E^u(t) \boxed{+} s)$ where $s = \mathit{symObj}(t, E^u)$, a fresh constant relation representing a fresh object. In this case, we define
$\bar{\delta^*} = \delta^* \cup \{s \mapsto \{obj\}\}$
Therefore, we have

$$[\![E^l_{post}(v)]\!]_i^{\bar{\delta^*}} \subseteq c(v) \subseteq [\![E^u_{post}(v)]\!]_i^{\bar{\delta^*}}$$

$$[\![E^l_{post}(t)]\!]_i^{\bar{\delta^*}} \subseteq c(t) \subseteq [\![E^u_{post}(t)]\!]_i^{\bar{\delta^*}}$$

Furthermore, since

$$\forall x \neq v, x \neq t \mid (c(x) = c^*(x)) \wedge (E^l_{post}(x) = E^l(x)) \wedge (E^u_{post}(x) = E^u(x))$$

we have

$$\forall x.\ [\![E^l_{post}(x)]\!]_i^{\bar{\delta^*}} \subseteq c(x) \subseteq [\![E^u_{post}(x)]\!]_i^{\bar{\delta^*}}$$

That is, $c \in \gamma(\langle E^l_{post},\ E^u_{post}\rangle)$, and the composition property holds.

Return statements. These statements are treated as assignments to special variables that store the return values of functions. Since the composition property holds for assignments, it holds for return statements too.

Branches. Let the statement S be **if** (*cond*) S_1 **else** S_2. We show that if the composition property holds for S_1 and S_2, it holds for S too. By the definition of $\mathcal{F}$ (Figure 4-13), we have

$$\mathcal{F}(\mathbf{if}(\mathit{cond})S_1\mathbf{else}S_2, \gamma(\langle E^l,\ E^u\rangle)) \subseteq \mathcal{F}(S_1, \gamma(\langle E^l,\ E^u\rangle)) \cup \mathcal{F}(S_2, \gamma(\langle E^l,\ E^u\rangle))$$

Let $\bar{\mathcal{F}}(S_1, \langle E^l,\ E^u\rangle) = \langle E^l_1,\ E^u_1\rangle$ and $\bar{\mathcal{F}}(S_2, \langle E^l,\ E^u\rangle) = \langle E^l_2,\ E^u_2\rangle$. Since the composition property holds for S_1 and S_2, we have

$$\mathcal{F}(S_1, \gamma(\langle E^l,\ E^u\rangle)) \subseteq \gamma(\langle E^l_1,\ E^u_1\rangle) \text{ and } \mathcal{F}(S_2, \gamma(\langle E^l,\ E^u\rangle)) \subseteq \gamma(\langle E^l_2,\ E^u_2\rangle)$$

and by Lemma 3, we have

$$\gamma(\langle E^l_1,\ E^u_1\rangle) \cup \gamma(\langle E^l_2,\ E^u_2\rangle) \subseteq \gamma(\langle E^l_1,\ E^u_1\rangle \sqcup \langle E^l_2,\ E^u_2\rangle)$$

Therefore, the composition property holds:

$$\mathcal{F}(\mathbf{if}(\mathit{cond})S_1\mathbf{else}S_2, \gamma(\langle E^l,\ E^u\rangle)) \subseteq \gamma(\bar{\mathcal{F}}(\mathbf{if}\ (\mathit{cond})S_1\mathbf{else}\ S_2, \langle E^l,\ E^u\rangle))$$

Loops. Let the statement S be **while** (*cond*) S_1. We show that if the composition property holds for S_1, it holds for S too. Since the widening operator is safe (Lemma 4), fixpoint computation is safe. Therefore, in order to prove the composition property of the loop abstraction, it is sufficient to show that the composition property holds after abstracting each loop iteration. That is,

$$\mathcal{F}(S_1, \gamma(\langle E^l,\ E^u\rangle)) \subseteq \gamma(\bar{\mathcal{F}}(S_1, \mathit{addCond}(\mathit{cond}, \langle E^l,\ E^u\rangle)))$$

Since the *addCond* function is safe (Lemma 5), all concrete states in which the condition *cond* evaluates to true belong to $\gamma(\mathit{addCond}(\mathit{cond}, \langle E^l,\ E^u\rangle))$. At each iteration, the loop statement S_1 is executed iff the loop condition is true. Therefore,

$$\mathcal{F}(S_1, \gamma(\langle E^l,\ E^u\rangle)) = \mathcal{F}(S_1, \gamma(\mathit{addCond}(\mathit{cond}, \langle E^l,\ E^u\rangle)))$$

Since the composition property holds for S_1, we have

$$\mathcal{F}(S_1, \gamma(\mathit{addCond}(\mathit{cond}, \langle E^l,\ E^u\rangle))) \subseteq \gamma(\bar{\mathcal{F}}(S_1, \mathit{addCond}(\mathit{cond}, \langle E^l,\ E^u\rangle)))$$

Therefore, the composition property holds for each loop iteration:

$$\mathcal{F}(S_1, \gamma(\langle E^l,\ E^u\rangle)) \subseteq \gamma(\bar{\mathcal{F}}(S_1, \mathit{addCond}(\mathit{cond}, \langle E^l,\ E^u\rangle)))$$

Call sites. We define a new transfer function $\bar{\mathcal{F}}_{\mathit{inline}}$ that summarizes the body of a procedure from scratch every time it is called.

$$\begin{cases} \bar{\mathcal{F}}_{\mathit{inline}}(S, \langle E^l,\ E^u\rangle) = \bar{\mathcal{F}}(S, \langle E^l,\ E^u\rangle) & \texttt{if}\ S \notin \mathit{CallStmt} \\ \bar{\mathcal{F}}_{\mathit{inline}}(\mathit{proc}(e_1, \ldots, e_n), \langle E^l,\ E^u\rangle) = & \\ \quad \bar{\mathcal{F}}_{\mathit{inline}}(\mathit{proc\_body}, \langle E^l[\mathit{forml}_1 \mapsto \mathit{eval}^l(e_1), \ldots, \mathit{forml}_n \mapsto \mathit{eval}^l(e_n)], & \\ \qquad\qquad E^u[\mathit{forml}_1 \mapsto \mathit{eval}^u(e_1), \ldots, \mathit{forml}_n \mapsto \mathit{eval}^u(e_n)]\rangle) & \end{cases}$$

That is, $\bar{\mathcal{F}}_{\mathit{inline}}$ summarizes procedure calls by inlining them. Therefore, it is easy to see that the composition property holds for $\bar{\mathcal{F}}_{\mathit{inline}}$.

The transfer function $\bar{\mathcal{F}}$, however, does not inline procedure calls. Instead, it summarizes a call by computing its calling context, extracting a template summary for that context (if it is not already available), and instantiating the template summary using the actual environment values.

In order to show that the composition property holds for $\bar{\mathcal{F}}$, we argue that $\bar{\mathcal{F}}$ generates the same results as $\bar{\mathcal{F}}_{\mathit{inline}}$. According to the definition of $\bar{\mathcal{F}}$, the abstraction of a statement with respect to a pair of environments does not depend on the actual values of the environments; it only depends on whether the values of the accessed fields, variables, or allocated objects are exact or not. This information is stored in the calling context. Thus, abstracting a procedure call with respect to its calling context will invoke the same abstraction rules as abstracting it with respect to the actual pair of environments at the call site. Therefore, the template summary generated by $\bar{\mathcal{F}}$ is the same as the summary generated by $\bar{\mathcal{F}}_{inline}$ except for relation names: the actual values of relations at the call site are replaced by fresh symbolic constants. The instantiation phase replaces these constants with their actual values. Therefore, the final summary returned by $\bar{\mathcal{F}}$ is exactly the same as the one returned by $\bar{\mathcal{F}}_{inline}$.

To avoid abstracting a procedure an exponential number of times, we may use the template summary of the empty context in other contexts. Because the empty context represents the case where none of the values are exact, it only invokes abstraction rules corresponding to weak updates. Therefore, although in this case the summary generated by $\bar{\mathcal{F}}$ is not the same as the one generated by $\bar{\mathcal{F}}_{inline}$, it is weaker. That is, $\gamma(\bar{\mathcal{F}}_{inline}(proc, \langle E^l,\ E^u\rangle)) \subseteq \gamma(\bar{\mathcal{F}}(proc, \langle E^l,\ E^u\rangle))$. Therefore, the composition property still holds for $\bar{\mathcal{F}}$.

Sequence of statements. Let the statement S be $S_1; S_2$. We show that if the composition property holds for S_1 and S_2, it holds for S too. By the composition property of S_1, we have

$$\mathcal{F}(S_1, \gamma(\langle E^l,\ E^u\rangle)) \subseteq \gamma(\bar{\mathcal{F}}(S_1, \langle E^l,\ E^u\rangle))$$

which implies that

$$\mathcal{F}(S_2, \mathcal{F}(S_1, \gamma(\langle E^l,\ E^u\rangle))) \subseteq \mathcal{F}(S_2, \gamma(\bar{\mathcal{F}}(S_1, \langle E^l,\ E^u\rangle)))$$

By the composition property of S_2, we have $\mathcal{F}(S_2, \gamma(\langle E^l,\ E^u\rangle)) \subseteq \gamma(\bar{\mathcal{F}}(S_2, \langle E^l,\ E^u\rangle))$ in which the arbitrary pair of $\langle E^l,\ E^u\rangle$ can be replaced by $\bar{\mathcal{F}}(S_1, \langle E^l,\ E^u\rangle)$:

$$\mathcal{F}(S_2, \gamma(\bar{\mathcal{F}}(S_1, \langle E^l,\ E^u\rangle))) \subseteq \gamma(\bar{\mathcal{F}}(S_2, \bar{\mathcal{F}}(S_1, \langle E^l,\ E^u\rangle)))$$

Therefore,

$$\mathcal{F}(S_2, \mathcal{F}(S_1, \gamma(\langle E^l,\ E^u\rangle))) \subseteq \gamma(\bar{\mathcal{F}}(S_2, \bar{\mathcal{F}}(S_1, \langle E^l,\ E^u\rangle)))$$

By the definitions of $\mathcal{F}$ and $\bar{\mathcal{F}}$, we have

$$\mathcal{F}(S_1; S_2, \gamma(\langle E^l,\ E^u\rangle)) \subseteq \gamma(\bar{\mathcal{F}}(S_1; S_2, \langle E^l,\ E^u\rangle))$$

That is, the composition property holds.

4.5.2 Termination

In order to prove that the abstraction algorithm always terminates, it is sufficient to show that the abstraction of all loops and recursive procedures reaches a fixpoint after a finite number of iterations.

Both least upper bound and widening operations result in a lattice point which is either the same as one of their arguments or one that is higher than both of them. During the abstraction of a loop, if the join operation results in a lattice point which is the same as one of its arguments, then it has reached a fixpoint and the abstraction terminates. If the join instead results in a node higher in the lattice, the abstraction can make only a finite number of moves up the lattice before the abstract environment is widened to $\top$. This is because the length of all expressions is bounded by a constant size, and thus the height of the lattice is finite. Therefore, after a finite number of steps, the abstraction will either reach a fixpoint at $\top$ or a fixpoint at some lower point. Thus, it always terminates.

4.5.3 Complexity

We prove that the complexity of our abstraction algorithm is bounded by $O(nm)$ where n denotes the size of the procedure being abstracted (including all of its reachable procedures) in a 3-address instruction format[3], and m denotes the total number of fields, types, global variables, and the maximum number of formal parameters of all reachable procedures.

We assume that looking up and updating a value in an environment has a constant cost. Since the length of any program expression in a 3-address instruction format

[3] Java programs can be converted to this format using the soot optimization package [57].

is at most 2, the cost of evaluating an expression using $eval^l$ or $eval^u$ is constant. Furthermore, simplifying the relational expressions using the simplification rules of Figure 4-11 has also a constant cost. This is because the length of a relational expression is bounded by a constant, and all simplification rules strictly reduce the length of the expression being simplified.

Since applying evaluation functions and simplification rules has a constant cost, abstracting simple statements (i.e. assignments, field updates, allocations, and return statements) has a constant cost. Abstracting a conditional involves abstracting its branches and merging the results by computing their least upper bound. The merge operation updates the values of all fields, variables, and types accessed by each branch. Therefore, its computation time is $O(m)$.

Abstracting a loop involves computing a fixpoint. Because the length of the generated relational expressions is bounded by a constant, computing a fixpoint needs at most a constant number of iterations. After each iteration, however, we have to compute the least upper bound of the current abstraction with the previous one and check whether a fixpoint is reached or not. This computation takes $O(m)$ time.

Abstracting a call site involves computing a template pair of environments based on the calling context, abstracting the body of the callee, and instantiating the template summary. This computation also takes $O(m)$ time because the values of all fields, types, and variables have to be updated. Since the total number of times that a specific procedure may be summarized is bounded by a constant, the cost of abstracting a procedure is at most $O(m)$.

During the abstraction, each statement is visited a constant number of times: simple statements and the statements that belong to conditional branches are visited once, and the statements that form the bodies of loops and called procedures are visited at most a pre-defined constant number of times. Since the cost of abstracting each statement is at most $O(m)$, the total abstraction cost is at most $O(mn)$.

4.6 Optimization

A common loop pattern in heap-manipulating programs is iterating over a linked data structure: a loop traverses over a data structure using a variable x that acts as a cursor. In each iteration of the loop, the cursor variable is updated by taking one step along a relation r; the termination condition is that no further step along r can be taken. In a linked list, for example, x is a reference to a list entry, r is the *next* field from entry to entry, and the termination condition is *x.next*!=*null*. In a traversal using an iterator, x is the reference to the iterator, r is the specification field associated with iteration, and the termination condition is *hasNext*(x).

The general form of the loop is:

```
while (cond(x, r)) {
   S1; x=next(x); S2; }
```

which allows arbitrary statements before and after the update of the cursor, so long as they do not mutate the relation r or assign to the cursor x.

For any loop in this form, our analysis treats the approximation of x by $x_0. * r$ (where x_0 is the initial value of x and $*r$ is the reflexive-transitive closure of r) as the exact set of values taken by x during the execution of the loop.

Having inferred the exact value of the loop variable, the analysis often generates more precise abstraction of the loop body by performing an additional optimization pass over the loop. This pass infers a more precise final value for any field which is (1) updated exactly once in the loop body and (2) the updating statement is of the form $e_1.f = e_2$ where e_1 has an exact value and e_2 is constant with respect to the loop. This additional precision will often carry through the abstraction rules.

Recall the example we saw in Figure 4-5:

```
void Graph.init() {
    ListEntry c = this.nodes.head;
    while (c != null) {
        c.node.visitedInsNum = 0;
        c = c.next;
    }
}
```

This loop matches the optimized pattern, so the exact value of c will be this.nodes.head.*next. After intersecting with loop condition, the exact value will be this.nodes.head.*next & (ListEntry' - null). Since the value of c is exact, and the right-hand side of the update statement is a constant, the abstraction of the statement c.node.visitedInsNum = 0 is also exact, giving us the following (after simplifications):

```
visitedInsNum' = visitedInsNum ++ (this.nodes.head.*next.node -> 0)
```

Without this optimization we would have generated the following:

```
visitedInsNum' ⊇ visitedInsNum - (this.nodes.head.*next.node -> Int)
visitedInsNum' ⊆ visitedInsNum + (this.nodes.head.*next.node -> 0)
```

which allows an arbitrary subset of this.nodes.head.*next.node.visitedInsNum to be overridden by 0, and thus, is a much weaker specification.

Chapter 5

Checking Abstract Programs: Forge

This chapter describes how a Java program is abstracted and checked against a given property. The abstraction involves translating the program's statements to an Alloy formula, using the initial specifications of its called procedures to encode the call sites. The resulting Alloy formula is then checked against the given property using an Alloy model finder. We use Forge [14], a bounded verification tool for Java programs, to translate Java programs to Alloy formulas and perform the analysis. In order to translate the code, Forge computes a symbolic state in which the values of variables and fields are encoded as relational expressions. While this approach produces a compact Alloy formula by avoiding intermediate relations, it cannot readily encode the kind of call site specifications that the technique of Chapter 4 extracts. Therefore, a special encoding is necessary to incorporate those specifications into the Forge translation. This chapter briefly describes the ideas underlying Forge, and explains how call site specifications are encoded.

5.1 Overview

At each iteration of our specification refinement technique, described in Chapter 2, the top-level procedure is checked against the given property, using the specifications

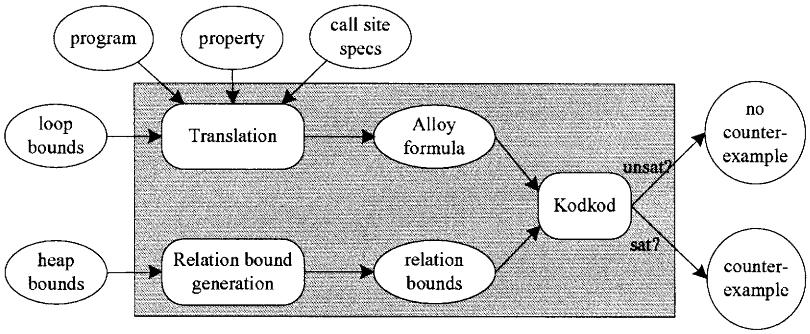


Figure 5-1: Forge architecture: the gray box shows Forge internal processes.

inferred for the call sites as surrogates for their code. We use Forge to perform this checking.

Figure 5-1 shows the architecture of Forge. It takes a Java program in which a procedure is selected for analysis, a property expressed in the Alloy language, and some bounds on the number of loop iterations and the size of the heap. Forge checks whether the selected procedure satisfies the given property or not. If so, it terminates with no counterexamples, meaning that the property holds within the checked bounds. Otherwise, it outputs a sound counterexample. Therefore, its analysis is categorized as bounded verification.

Forge analyzes programs in a modular way. If the user chooses to provide a specification for a procedure, that specification will replace the body of that procedure anywhere it is called. Otherwise, the calls to that procedure will be inlined. Since our specification refinement technique infers the specifications of the call sites, we always use Forge in the former mode, that is by providing the specifications of the call sites.

Forge analyzes a procedure by generating an Alloy formula that encodes both the procedure and the negation of the given property, and solving it using Kodkod [56], a model finder for Alloy formulas. Kodkod requires finite lower and upper bounds on the values of the relations used in the analyzed Alloy formula. Thus, Forge includes a separate phase that produces such bounds based on the user-provided bounds on the size of the heap.

Kodkod translates the given Alloy formula to a boolean satisfiability formula in conjunctive normal form (CNF), and solves it using an off-the-shelf SAT solver. If a solution is found, Kodkod translates it back to some valid values for the relations in the Alloy formula and returns it to the user. Using Kodkod as an intermediate engine for translating Java programs to boolean formulas enables us to take advantage of the optimizations such as sharing detection and symmetry breaking offered by Kodkod.

The rest of this chapter briefly describes how Forge and Kodkod perform different phases of the analysis. More details can be found elsewhere [14, 56].

5.2 Translating Java to Alloy

A Java procedure can be encoded as a conjunction of Alloy formulas whose satisfying solutions denote executions of the procedure. This encoding is based on a relational view of the heap, previously described in Chapter 4: a field f of type T defined in a class C is viewed as a binary functional relation $f : C$ `->` T that maps each object of type C to at most one object of type T. Local variables and arguments are viewed as singleton sets, i.e. sets containing only one element.

Prior to translating the body of a procedure to Alloy, Forge uses the Soot framework [57] to transform the code to a 3-address instruction format in which all expressions are side-effect free. Furthermore, it performs the following transformations in a pre-process phase:

- All loops reachable from the analyzed procedure are unrolled based on some user-provided bound. Unrolling a loop produces a nested `if` statement which ends with an `assume` statement that forces the loop condition to be false. For example, unrolling a loop **while** `(cond) S;` twice results in
  if `(cond)` **then** `{S;` **if** `(cond)` **then** `{S;` **assume** `(not cond);}}`
  Recursive procedure calls are unrolled similarly.

- All call sites that can be dynamically dispatched are expanded to statically invoke all procedures that may be called at that site. The expansion produces a nested `if` statement that tests the type of the receiver object and calls the appropriate procedure accordingly.

- All call sites whose specifications are not available are inlined.

5.2.1 Encoding Basic Statements

In order to translate the body of a procedure to an Alloy formula, Forge computes a *symbolic state* and a *constraint* at each control point of the procedure. The state maps each variable and field to a relational expression that encodes its value, and the constraint accumulates all constraints on those expressions, generated by translating branches, allocations, and assume statements.

Given a pre-state, each statement is executed symbolically to produce a post-state and a constraint that encode the effects of that statement. When the symbolic execution of a procedure terminates, the post-state gives the final values of all fields and variables as expressions over their initial values. Forge then substitutes these expressions for the fields and variables that are mentioned in the property of interest, negates the resulting formula, conjoins it with the constraint produced during the symbolic execution, and hands it to Kodkod for a solution. A solution therefore gives a valid execution of the code that violates the property.

The translation starts with declaring a unary relation (corresponding to an Alloy type) for each class declared in the analyzed program. The value of this relation represents the set of all objects of that class that can exist during the lifetime of the program. Furthermore, an initial state is constructed in which an Alloy relation is declared for each field and formal parameter accessible by the analyzed procedure. The values of these relations represent the values of the fields and formal parameters in the pre-state of the procedure.

Given a pre-state s, each statement *stmt* produces a post-state $post(stmt, s)$ and a constraint $constr(stmt, s)$ according to the following rules:

Sequence of statements. A sequence of two statements $stmt_1; stmt_2$ is encoded by first encoding the first statement $stmt_1$ on the pre-state, and then encoding the second statement $stmt_2$ on the resulting state. The constraints generated by these encodings will be conjoined. That is,

$$\begin{aligned}&post(stmt_1; stmt_2, s) = post(stmt_2, post(stmt_1, s))\\&constr(stmt_1; stmt_2, s) = constr(stmt_1, s)\ \textbf{and}\ constr(stmt_2, post(stmt_1, s))\end{aligned}$$

Assignments to locals. An assignment statement $v = e$ is encoded by first evaluating the expression e in the pre-state s, and then storing it as the value of the variable v in the post-state. That is,

$$\begin{aligned}&post(v = e, s) = s[v \mapsto eval(e, s)]\\&constr(v = e, s) = true\end{aligned}$$

where $eval(e, s)$ evaluates a program expression e in a state s.

Field updates. A field update statement $e_1.f = e_2$ is concisely encoded using a relational override:

$$\begin{aligned}&post(e_1.f = e_2, s) = s[f \mapsto eval(f, s)\ \texttt{++}\ (eval(e_1, s)\ \texttt{->}\ eval(e_2, s))]\\&constr(e_1.f = e_2, s) = true\end{aligned}$$

That is, the value of f in the post-state is the same as its value in the pre-state except that it maps the value of e_1 to the value of e_2.

Conditionals. A conditional is encoded by first encoding its branches separately, and then merging the results by forming an Alloy `if` expression for every variable and field mutated by either branch. A conditional statement

if $(cond)$ $stmt_1$; **else** $stmt_2$;

is encoded as follows:

$$\begin{aligned}&post(\textbf{if}\ (cond)\ stmt_1;\ \textbf{else}\ stmt_2, s) =\\&\quad \textbf{let}\ s'_1 = post(stmt_1, s), s'_2 = post(stmt_2, s)\ \textbf{in}\\&\quad \lambda x.(\textbf{if}\ eval(cond, s)\ \textbf{then}\ eval(x, s'_1)\ \textbf{else}\ eval(x, s'_2))\\&constr(\textbf{if}\ (cond)\ stmt_1;\ \textbf{else}\ stmt_2, s) =\\&\quad \textbf{let}\ c'_1 = constr(stmt_1, s), c'_2 = constr(stmt_2, s)\ \textbf{in}\\&\quad (eval(cond, s)\ \texttt{=>}\ c'_1)\ \textbf{and}\ ((\textbf{not}\ eval(cond, s))\ \texttt{=>}\ c'_2)\end{aligned}$$

This encoding ensures that the final value of each variable and field in the post-state is its value in the `then` branch if the branch condition evaluates to true, and the one in the `else` branch otherwise. Furthermore, if the branch condition evaluates to true, the constraints produced by the `then` branch will be in effect. Otherwise, those produced by the `else` branch are effective.

Allocations. In order to encode allocation statements, we need a notion of *fresh objects* that can be allocated at each control point of the code. For each datatype in the program, we can introduce a set (unary relation) to represent fresh objects of that type. Allocating an object will then be encoded by choosing an element of this set non-deterministically and updating the set to exclude the chosen element.

This non-deterministic approach, however, results in an Alloy formula with many symmetric solutions, and thus is not efficient in practice. Therefore, in order to exploit the symmetry breaking feature of Kodkod, Forge determinizes the order in which fresh objects are allocated. That is, for each datatype t, Forge introduces a new relation *t_ord* that totally orders all the elements of type t. This ordering denotes the order in which new objects of type t are allocated during the execution of the analyzed procedure. The expression *first*(*t_ord*) denotes the first element of the *t_ord* ordering. Furthermore, Forge augments the symbolic state to include a mapping from each class to an expression that represents its last instantiated object.

Allocation statements are encoded by constructing an expression for the next allocated object and storing it in the post-state. An allocation statement $v = \textbf{new}\ t$ is therefore encoded as

$$\begin{array}{l}
\textbf{let}\ \mathit{lastObj} = \mathit{eval}(t, s), \\
\quad \mathit{newObj} = \textbf{if}\ (\textbf{no}\ \mathit{lastObj})\ \textbf{then}\ \mathit{first}(\mathit{t\_ord})\ \textbf{else}\ \mathit{lastObj.t\_ord}\ \ \textbf{in} \\
\ \ \mathit{post}(v = \textbf{new}\ t, s) = s[v \mapsto \mathit{newObj}, t \mapsto \mathit{newObj}] \\
\ \ \mathit{constr}(v = \textbf{new}\ t, s) = (\textbf{some}\ \mathit{newObj})
\end{array}$$

That is, if no objects of type t are previously allocated, the object instantiated by this statement will be the first object in the total ordering. Otherwise, it will be the object immediately following the last allocated object. This expression will be stored in the post-state both as the value of the variable v, and as the last allocated object of type t. Furthermore, we have to ensure that this new object exists. Therefore, the constraint forces the new object to have some value.

Assume statements. These statements are generated by the loop unrolling phase. A statement **assume** (*cond*) is encoded as follows:

$$\begin{aligned}
&\mathit{post}(\textbf{assume }(\mathit{cond}), s) = s\\
&\mathit{contr}(\textbf{assume }(\mathit{cond}), s) = \mathit{eval}(\mathit{cond}, s)
\end{aligned}$$

That is, the constraint ensures that the assume condition holds.

Return statements. The return statement **return** e in a procedure *proc* is treated as an assignment to a special variable *proc$return* that stores the return value of *proc*. That is,

$$\begin{aligned}
&\mathit{post}(\textbf{return } e, s) = s[\mathit{proc\$return} \mapsto \mathit{eval}(e, s)]\\
&\mathit{constr}(\textbf{return } e, s) = \mathit{true}
\end{aligned}$$

5.2.2 Encoding Call Site Specifications

Although Forge does not require users to provide specifications for called procedures, its modular analysis enables it to exploit such specifications whenever available. As mentioned before, we always provide Forge with the call site specifications that are inferred by our specification refinement technique. In this section, we describe how initial specifications are incorporated into the Forge framework.

As described in Chapter 4, the initial specification of a procedure is a formula that provides lower and upper bounds for the procedure's variables, fields, and types. The first step in using the specification of a called procedure is to substitute the actual values used at the call site (represented by Alloy expressions) for the symbolic names that are used in the specification. The second step is to generate a post-state and a constraint that encode the specification. These steps are described below.

Replacing symbolic names. A specification extracted by the technique of Chapter 4 contains two kinds of symbolic names that must be substituted: (1) *pre-*

state names that represent the values of relations (formal parameters and fields) in the pre-state, and (2) *object names* that represent allocated objects. Therefore, in order to replace symbolic names, two mappings are required:

$$\mathit{prestateMap} : \mathit{Names} \longrightarrow \mathit{RelationalExprs}$$

that maps the pre-state names used in the specification to the Alloy expressions that represent the actual values of relations in the pre-state of the call site, and

$$\mathit{objectsMap} : \mathit{Names} \longrightarrow \mathit{RelationalExprs}$$

that maps the object names used in the specification to some Alloy expressions that represent fresh objects. Given these two mappings, the *substitute* function substitutes the given Alloy expressions for their corresponding symbolic names:

$$\begin{aligned}
&\mathit{Mapping} = \mathit{Names} \longrightarrow \mathit{RelationalExprs}\\
&\mathit{substitute} : (\mathit{Formula} \times \mathit{Mapping} \times \mathit{Mapping}) \longrightarrow \mathit{Formula}\\
&\mathit{substitute}(\mathit{spec}, \mathit{prestateMap}, \mathit{objectsMap}) =\\
&\quad \mathit{spec}[p_i \longrightarrow \mathit{prestateMap}(p_i)][o_j \longrightarrow \mathit{objectsMap}(o_j)]\\
&\qquad\qquad\qquad\qquad\qquad\qquad \forall p_i \in \mathit{PrestateNames},\ o_j \in \mathit{ObjectNames}
\end{aligned}$$

where $f[n \longrightarrow e]$ substitutes an expression e for all occurrences of a name n in a formula f. Computing *prestateMap* is simple: given the call site's pre-state s, the mapping *prestateMap* maps each field and formal parameter of the called procedure to its actual value in s.

Computing *objectsMap* involves producing Alloy expressions that represent objects that are not already allocated. To produce such expressions, we follow the Forge framework: for each datatype t, we keep track of the last object allocated of type t – denoted by an auxiliary variable $\mathit{lastObj}_t$ – and the constraints necessary for allocating objects of type t – denoted by an auxiliary variable constr_t. Given the call site's pre-state s, the variable $\mathit{lastObj}_t$ is initialized to the value $\mathit{eval}(t, s)$ and constr_t is initialized to the formula true. The mapping *objectsMap* is then computed as follows:

- *Lower bound.* As explained in Chapter 4, for each datatype t, the initial specification of a procedure provides a lower bound that represents the objects of type t that are allocated in every execution of the procedure. For a symbolic object *symObj* in the lower bound of t, we compute *objectsMap*(*symObj*) as follows:

$$\begin{array}{l}\textbf{let } \mathit{newObj} = \textbf{if } (\textbf{no } \mathit{lastObj}_t) \textbf{ then } \mathit{first}(\mathit{t\_ord}) \textbf{ else } \mathit{lastObj}_t.\mathit{t\_ord} \textbf{ in}\\ \quad \mathit{objectsMap}(\mathit{symObj}) = \mathit{newObj}\\ \quad \mathit{constr}_t = \mathit{constr}_t \textbf{ and } (\textbf{some } \mathit{newObj})\\ \quad \mathit{lastObj}_t = \mathit{newObj}\end{array}$$

That is, *symObj* is mapped to *newObj* which is an Alloy expression representing the next available object to allocate. The formula constr_t is updated to ensure that this object exists, and $\mathit{lastObj}_t$ is updated to represent that this expression is the last allocated object of type t. These updated values of constr_t and $\mathit{lastObj}_t$ are then used to compute the Alloy expressions of the next symbolic object in the lower bound of t. This process continues until all symbolic objects in the lower bound of t are mapped to Alloy expressions.

- *Upper bound (individual objects).* As explained in Chapter 4, for each datatype t, the initial specification of a procedure provides an upper bound that represents the objects that may be allocated in some execution of the procedure. For a symbolic object *symObj* in the upper bound of t that is not already mapped to any Alloy expression (that is, it does not appear in the lower bound of t), we introduce a fresh unary relation r whose value represents the fresh object corresponding to *symObj*:

$$\begin{array}{l}\textbf{let } \mathit{newObj} = \textbf{if } (\textbf{no } \mathit{lastObj}_t) \textbf{ then } \mathit{first}(\mathit{t\_ord}) \textbf{ else } \mathit{lastObj}_t.\mathit{t\_ord} \textbf{ in}\\ \quad \mathit{objectsMap}(\mathit{symObj}) = r\\ \quad \mathit{constr}_t = \mathit{constr}_t \textbf{ and } ((\textbf{no } r) \textbf{ or } (r = \mathit{newObj}))\\ \quad \mathit{lastObj}_t = \textbf{if } (\textbf{no } r) \textbf{ then } \mathit{lastObj}_t \textbf{ else } r\end{array}$$

The constraint specifies that r can be either empty (representing the case that this object is not allocated), or *newObj* (the next object that can be instantiated). The variable $lastObj_t$ is updated to correctly represent the last allocated object in either case. The updated $constr_t$ and $lastObj_t$ are used to compute the mapping of the next symbolic object in the upper bound of t. This process continues until all symbolic objects in the upper bound of t are mapped to Alloy expressions.

- *Upper bound (set of objects).* If the upper bound of t contains a symbolic set of allocated objects, a fresh unary relation r is introduced to represent the set of objects abstracted by this symbolic set:

```
let $newObj =$ if (no $lastObj_t$) then $first(t\_ord)$ else $lastObj_t.t\_ord$ in
    $objectsMap(symObjSet) = r$
    $constr_t = constr_t$ and ($r$ in $newObj.*t\_ord$)
    $lastObj_t =$ if (no $r$) then $lastObj_t$
                    else ($r$ - $r.$^~$t\_ord$)
```

  That is, r can contain any number of objects that are not instantiated yet (i.e. they belong to $newObj.*t\_ord$). The $lastObj_t$ variable is then updated to store the last object contained in r.

Having computed *objectsMap* and *prestateMap*, we use the *substitute* function to substitute the computed Alloy expression for every symbolic name that is used in the specification of the called procedure. The resulting specification is then used to compute the post-state and the necessary constraints of the call site.

Computing post-state. Given a call site $p(e_1, .., e_n)$ and a pre-state s, let $spec'$ denote the specification of p after the substitution of symbolic names. This specification provides upper and lower bounds for p's accessed types, fields, and the return value. In this section, we describe how these bounds are used to compute the post-state of the call site.

- Types. The post-state of the call site maps each datatype t to an Alloy expression that represents the last allocated object of type t. This expression is computed while the mapping *objectsMap* is computed, and is stored in the auxiliary variable $lastObj_t$. Therefore, $lastObj_t$ gives the post-state of a type t.

$$post(p(e_1, .., e_n), s)(t) = lastObj_t \qquad\qquad \forall t \in Type$$

- Fields. The specification constrains the final value of each field f by the two constraints ($f \subseteq ub$) and ($f \supseteq lb$) where ub and lb are Alloy expressions representing the upper and lower bounds of the relation corresponding to f. Let $constr_f$ denote the constraint generated to encode the post-state of f. The following two cases are distinguished:

  (1) If ($ub = lb$), meaning that the expression extracted for f is exact, then this expression is stored in the post-state. That is,

  $$\begin{aligned}&post(p(e_1, .., e_n), s)(f) = ub\\ &constr_f = true\end{aligned}$$

  (2) Otherwise, the exact final value of f is unknown. Therefore, a fresh unary relation r is allocated to represent this value. It is stored in the post-state, and constrained as follows:

  $$\begin{aligned}&post(p(e_1, .., e_n), s)(f) = r\\ &constr_f = (r \textbf{ in } ub) \textbf{ and } (lb \textbf{ in } r)\end{aligned}$$

  That is, f can take any arbitrary value in the post-state as long as it conforms to the lower and upper bounds.

- Return value. The specification constrains the return value of p by the two constraints ($p\$return \subseteq ub$) and ($p\$return \supseteq lb$). These constraints are encoded similarly to the above case. That is, if the lower and upper bounds are the same, they are used as the value of *p$return* in the post-state. Otherwise, an unconstrained fresh binary relation r is allocated to represent the value of *p$return* in the post-state. A constraint $constr_{ret}$ bounds the value of this

```
void Graph.init():
  visitedInsNum' = visitedInsNum ++ (this.nodes.head.*next.node -> 0)

void Graph.fixIns(Node n):
  visitedInsNum' ⊆ visitedInsNum + (n.outNodesList.head.*next.node -> Int)
  visitedInsNum' ⊇ visitedInsNum - (n.outNodesList.head.*next.node -> Int)

ListEntry Graph.findSource(ListEntry entry):
  findSource$return ⊆ (entry.*next &
                       (null + node.inDeg.(entry.*next.node.visitedInsNum)))
  findSource$return ⊇ none
```

Figure 5-2: Initial specifications of the called procedures.

relation by *lb* and *ub*.

The constraint encoding this call site accumulates all constraints generated for type allocations, fields, and the return value. That is,

$$constr(p(e_1, .., e_n), s) = constr_t \textbf{ and } constr_f \textbf{ and } constr_{ret} \qquad \forall t \in Type, f \in Field$$

5.2.3 Example

We illustrate the translation of Java procedures to Alloy formulas using the topological sort example of Chapter 1 (given again in Figure 5-3(a)). We use the initial specifications previously described in Chapter 4 to translate the call sites of the `topologicalSort` method. Those specifications are given again in Figure 5-2.

We assume that the loop in the `topologicalSort` method is unrolled once. The resulting code is given in Figure 5-3(a). It resolves the `break` statement in the loop body by adding an `else` branch to the inner `if` statement.

Figure 5-3(b) gives the Alloy encoding of this method as generated by Forge. It gives the symbolic state generated after translating each statement. However, to simplify the representation, instead of giving the complete state, this figure only shows the updated fields and local variables using some intermediate names with numerical subscripts. The values of the accessed fields and formal parameters in the pre- and post-state are denoted by unprimed and primed names, respectively. The fields not specified in the post-state are not modified by the method. The line numbers used in the encoding of Figure 5-3(b) correspond to the line numbers of the `topologicalSort` code of Figure 5-3(a).

Figure 5-3(c) gives the definition of *constr*, the conjunction of the constraints generated during the translation. We assume that the constraints of this figure are implicitly conjoined. In this figure, the line numbers preceding each constraint shows the line of the code that is responsible for generating that constraint.

The encoding of Figure 5-3(b) shows how each statement is translated. An assignment statement updates the value of the left-hand-side variable in the symbolic state to be equal to the right-hand-side value. An `if` statement is translated by first translating each branch, and then forming an Alloy `if` expression on the results at the merging point of the branches (Lines 15 and 16). An **`assume`** statement does not update the symbolic state; it augments the constraint to ensure that the assume condition holds (Line 14).

Call sites of the `topologicalSort` method are encoded initially using the specifications of Figure 5-2. The specification of the `init` method is accurate. Therefore, the call to this method is encoded by simply updating the value of the modified field `visitedInsNum` in the symbolic state (Line 2). The specifications of `findSource` and `fixIns`, however, are approximate. Therefore, the translations of the calls to these methods allocate fresh relations `source$1` and `visitedInsNum$1`, and augment the constraint to ensure that the values of these relations are bounded by the upper and lower bounds of the specification (Lines 5 and 9).

The encoding of Figure 5-3(b) is represented as a `let` statement. It substitutes the post-state expressions computed for the variables and fields in the formulas *constr* and (**not** *property*), where *constr* denotes the constraints generated during the translation (given in Figure 5-3(c)) and *property* denotes the property being analyzed. Therefore, any solution to this final formula represents an execution of the abstract code that violates the property.

```
an Graph.topologicalSort() {
oolean isAcyclic = true;
t();
stEntry cur = nodes.head;
 (cur != null) {
ListEntry source =
  findSource(cur);
if (source == null) {
  isAcyclic = false;
} else {
  fixIns(source.node);
  Node tmp = source.node;
  source.node = cur.node;
  cur.node = tmp;
  cur = cur.next;
  assume (cur == null)
}

turn isAcyclic;
```

```
1: let isAcyclic_1 = true_rel,
2:   visitedInsNum_1 = visitedInsNum ++
     (this.nodes.head.*next.node -> 0),
3:   cur_1 = this.nodes.head,
5:   source_1 = source$1,
7:   isAcyclic_2 = false_rel,
9:   visitedInsNum_2 = visitedInsNum$1,
10:  tmp_1 = source_1.node,
11:  node_1 = node ++ (source_1 -> cur_1.node),
12:  node_2 = node_1 ++ (cur_1 -> tmp_1),
13:  cur_2 = cur_1.next,
15:  isAcyclic_3 = if (source_1 = null_rel)
       then isAcyclic_2 else isAcyclic_1,
     visitedInsNum_3 = if (source_1 = null_rel)
       then visitedInsNum_1 else visitedInsNum_2,
     node_3 = if (source_1 = null_rel)
       then node else node_2,
     cur_3 = if (source_1 = null_rel)
       then cur_1 else cur_2,
     isAcyclic' = if (cur_1 != null_rel)
       then isAcyclic_3 else isAcyclic_1,
16:  visitedInsNum' = if (cur_1 != null_rel)
       then visitedInsNum_3 else visitedInsNum_1,
     node' = if (cur_1 != null_rel)
       then node_3 else node,
     cur' = if (cur_1 != null_rel)
       then cur_3 else cur_1,
17:  topologicalSort$return' = isAcyclic' in
     constr and (not property)
```

```
constr =

5: (cur_1 != null_rel) =>
   source$1 in (cur_1.*next & (null_rel +
    node.inDeg.(cur_1.*next.node.visitedInsNum_1)))
9: (cur_1 != null_rel) => (source_1 != null_rel) =>
   {(visitedInsNum$1 in visitedInsNum_1 +
    source_1.node.outNodesList.head.*next.node -> Int)
   (visitedInsNum_1 -
    source_1.node.outNodesList.head.*next.node->Int
    in visitedInsNum$1)}
14: (cur_1 != null_rel) => (source_1 != null_rel) =>
   (cur_2 = null_rel)
```

Figure 5-3: Translation example: (a) code with loop unrolled once, (b) Alloy encoding (c) generated constraints.

5.3 Generating Relation Bounds

In order to solve an Alloy formula, Kodkod, the Alloy model finder underlying Forge, requires a universe of atoms, a lower bound for each relation representing the tuples that must be included in that relation, and an upper bound for each relation representing the tuples that may be included in that relation. That is, in every solution found by Kodkod, the tuples of a relation must include all tuples in its lower bound, and must be chosen from the tuples in its upper bound. These bounds not only specify the scope of the space being analyzed, but also facilitate defining a partial solution. That is, if the exact tuples of some relation are known in advance, they can be provided to Kodkod as both the lower and the upper bound of that relation, thus ensuring that the value of that relation in all found solutions will be the set of those tuples.

Forge constructs relation bounds based on the user-provided bounds on the size of the heap: for each class C declared in the analyzed program, Forge requires the users to provide a number n_c denoting the maximum number of objects of type C that can exist in the analyzed heap. It uses a set of n_c fresh atoms as both the upper and the lower bound of the relation corresponding to C. Therefore, it ensures that Kodkod considers n_c objects of type C in the analysis.

The `Int` type is bounded by the user-provided number for the bitwidth of the integers. Given a bitwidth i, Forge constructs an atom for each number in the range of -2^{i-1} to $2^{i-1} - 1$. A set containing all those atoms is used as both the upper and the lower bound of the `Int` type.

The lower bound of each relation corresponding to a variable v of type C is an empty set. Its upper bound is the union of the set representing the upper bound of type C and the one representing the null relation.

Similarly, the lower bound of a binary relation r : C `->` T is an empty set, and its upper bound is the Cartesian product of the upper bound of C with the union of the upper bound of T and the null relation.

The universe of atoms is the union of sets of atoms that are constructed for each type.

5.4 Solving Alloy Formulas: Kodkod

Given an Alloy formula, a universe of atoms, and the bounds on the values of the relations used in that formula, Kodkod analyzes the formula for a solution within the given bounds. This analysis is performed by first converting the Alloy formula to a boolean formula, and then solving it using a SAT solver.

Alloy formulas are converted to boolean using the following basic idea [29]: Each n-ary relation r with a lower bound of b_L and an upper bound of b_U over a universe $U = \{a_0, \ldots, a_k\}$ is encoded as an n-dimensional boolean matrix m defined as follows:

$$m[i_1, \ldots, i_n] = \begin{cases} \textit{true} & \langle a_{i_1}, \ldots, a_{i_n} \rangle \in b_L \\ \textit{false} & \langle a_{i_1}, \ldots, a_{i_n} \rangle \notin b_U \\ \textit{fresh}() & \langle a_{i_1}, \ldots, a_{i_n} \rangle \in b_U - b_L \end{cases}$$

That is, any tuple in the lower bound of r must be included in m, any tuple not in the upper bound of r should not be included in m, and any tuple in the upper bound that is not in the lower bound of r is allocated a fresh boolean variable whose truth value will indicate whether that tuple is included in m or not.

These boolean matrices are then combined using matrix operations to represent different Alloy expressions. Alloy formulas are encoded by combining these matrices into propositional formulas. Kodkod then converts the propositional formulas to CNF using the standard translation from boolean logic to conjunctive normal form (see, for example, [17]).

In order to generate smaller boolean formulas, Kodkod exploits some optimization techniques such as symmetry breaking and sharing detection whenever possible. Details of the Kodkod translation can be found elsewhere [56].

The generated CNF formula is delegated to a SAT solver for a solution. A solution returned by the SAT solver assigns truth values to all boolean variables. Kodkod translates these values back to some symbolic values for the relations in the solved

Alloy formula. This is done using the mapping of the Alloy relations to boolean variables that was constructed by Kodkod during the translation phase. Kodkod then returns the Alloy solution as its output.

Because the formula generated by Forge encodes call sites using their initial over-approximating specifications, any solution found by Kodkod at this stage is a counterexample that conforms to the initial specifications of the call sites, rather than their actual code. Therefore, the validity of any found counterexample must be checked against the original code before returning to the user. The details of this process are explained in the next chapter.

Chapter 6

Refining Specifications

In each iteration of our specification refinement technique, an abstraction of a given procedure is checked against a given property. If a counterexample is found, it is checked for validity in the original procedure. If valid, the counterexample will be returned to the user. Otherwise, it will be used to refine the specification of some called procedure so that this invalid counterexample is eliminated. This chapter describes how a counterexample is checked for validity and how it is used to refine a specification. Validity checking involves checking the original code of all call sites against the counterexample in a hierarchical way. Refining a specification involves converting a proof of unsatisfiability generated by the underlying solver to a small well-formed Alloy formula that eliminates the invalid counterexample. The details of these phases are described in this chapter.

6.1 Validity Checking of Counterexamples

A counterexample is an execution of the analyzed program that violates the analyzed property. The only statements that may be abstract in an analyzed program are its procedure calls. Therefore, in order to check the validity of a counterexample, it is sufficient to only check the validity of the state transitions that the counterexample assigns to the call sites. This process is described in this section and illustrated by the topological sort example of Chapter 1.

```
void Graph.init():
  visitedInsNum' = visitedInsNum ++ (this.nodes.head.*next.node -> 0)

void Graph.fixIns(Node n):
  visitedInsNum' ⊆ visitedInsNum + (n.outNodesList.head.*next.node -> Int)
  visitedInsNum' ⊇ visitedInsNum - (n.outNodesList.head.*next.node -> Int)

ListEntry Graph.findSource(ListEntry entry):
  findSource$return ⊆ (entry.*next &
                         (null + node.inDeg.(entry.*next.node.visitedInsNum)))
  findSource$return ⊇ none
```

Figure 6-1: Initial specifications.

6.1.1 Interpretation of a Counterexample

As explained in Chapter 5, we use Forge to check an abstract procedure against a given property. Forge performs this checking by translating the procedure to a relational formula in Alloy, and solving it using Kodkod. Any counterexample returned by Forge will be an assignment of tuples to the relations that are mentioned in the solved Alloy formula, representing atoms by symbolic names. We use the values of relations to compute a state transition for every statement of the analyzed procedure, and thus construct a symbolic execution.

The relations in an Alloy formula produced by Forge are introduced at either of the following places: (1) at the beginning of the procedure to encode the values of the fields and formal parameters in the pre-state, and (2) at an approximate call site to encode the values of the fields and variables that are not precisely given by the call site specification. All other values are expressed by Alloy expressions over these relations. Therefore, a counterexample defines a pre-state for the analyzed procedure and post-states for its approximate call sites, and is sufficient to compute a symbolic execution of the procedure: starting from the pre-state, each statement of the code can be executed symbolically to produce the next program state. Approximate procedure calls, however, cannot be executed symbolically because their specifications are only approximations of their actual behaviors. The post-states of such call sites are therefore extracted from the counterexample.

```
/*
((repInv = true$rel) and
 (topologicalSort$return = true$rel)) =>
    (all e: this.nodes'.head'.*next' |
           int(e.node'.inDeg) < size(e.*prev')))
*/
boolean Graph.topologicalSort() {
1:  boolean isAcyclic = true;
2:  init();
3:  ListEntry cur = nodes.head;
4:  if (cur != null) {
5:    ListEntry source = findSource(cur);
6:    if (source == null) {
7:      isAcyclic = false;
8:    } else {
9:      fixIns(source.node);
10:     Node tmp = source.node;
11:     source.node = cur.node;
12:     cur.node = tmp;
13:     cur = cur.next;
14:     if (cur != null) {
15:       source = findSource(cur);
16:       if (source == null) {
17:         isAcyclic = false;
18:       } else {
19:         fixIns(source.node);
20:         tmp = source.node;
21:         source.node = cur.node;
22:         cur.node = tmp;
23:         cur = cur.next;
24:         assume (cur == null)
25:       }
26:     }
27:   }
28: }
29: return isAcyclic;
}
```

(a)

```
0: this, nodes, head, next, prev, node,
inDeg, outDeg, inNodesList,
outNodesList, visitedInsNum

5: source$1

9: visitedInsNum$1

15: source$2

19: visitedInsNum$2
```

(b)

Figure 6-2: Topological sort example: (a) Java code with 2 loop unrollings, (b) allocated Alloy relations.

Example. To illustrate how a counterexample gives an execution of a program, we use the topological sort example of Chapter 1. We check the `topologicalSort` method against the in-degree property discussed in that chapter. The property says that if the algorithm terminates successfully, the in-degree of each node is at most equal to the number of nodes preceding it in the final topological order. We analyze the code with respect to a heap containing 2 objects of each type, integers within a bitwidth of 3, and 2 unrollings of each loop. The initial specifications used to abstract the called procedures are given again in Figure 6-1.

Figure 6-2(a) gives the `topologicalSort` method after the loop unrollings, along with the property of interest. Translating this method to Alloy generates the relations shown in Figure 6-2(b). The line numbers in this figure show the lines of the code whose translations have generated the relations. Line 0 is the pre-state and other lines are the approximate call sites (calls to `findSource` and `fixIns`). Since the specification of the `init` method is accurate, the call to this method does not produce any new relations. The actual translation of the `topologicalSort` method to Alloy is similar to the one previously described in Section 5.2.3 except that it unrolls the loop twice rather than once. Hence, it is not shown here again.

Analyzing the `topologicalSort` method against the in-degree property results in the counterexample of Figure 6-3. Although this counterexample satisfies the initial specifications of the call sites, it is not valid with respect to their original code. Thus, it will be eliminated by the specification refinement phase as discussed in the next section.

As shown in Figure 6-3(a), the counterexample uses symbolic atom names to represent tuples of the Alloy relations that were produced during the translation. The value `G0` is an atom of type graph that denotes the receiver graph. The values `L0` and `L1` are list atoms. The values `E0` and `E1` are `ListEntry` atoms, and the values `N0` and `N1` are node atoms. The pre-state represents the heap given in Figure 6-3(b), and corresponds to the directed graph of Figure 6-3(c).

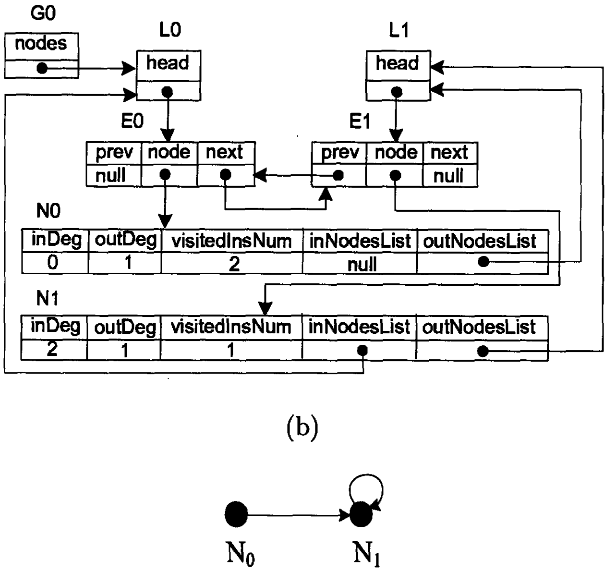
Figure 6-4 gives the symbolic program execution corresponding to this counterexample. It highlights the executed statements and annotates them by the values of pre-state relations:

(this = G0) (G0.nodes = L0) (L0.head = E0) (L1.head = E1)
(E0.next = E1) (E0.prev = null) (E0.node = N0)
(E1.next = null) (E1.prev = E0) (E1.node = N1)
(N0.inDeg = 0) (N0.outDeg = 1) (N0.inNodesList = null) (N0.outNodesList = L1)
(N0.visitedInsNum = 2)
(N1.inDeg = 2) (N1.outDeg = 1) (N1.inNodesList = L0) (N1.outNodesList = L1)
(N1.visitedInsNum = 1)

intermediate relations:

(source$1 = E0) (N0.visitedInsNum$1 = 0) (N1.visitedInsNum$1 = 2)
(source$2 = E1) (N0.visitedInsNum$2 = 0) (N1.visitedInsNum$2 = 1)

(a)



(b)

(c)

Figure 6-3: Counterexample: (a) textual description, (b) pre-state heap, (c) corresponding graph.

```
boolean Graph.topologicalSort() {
[(this = G0), (G0.nodes = L0), (L0.head = E0), (L1.head = E1), (E0.next = E1),
(E1.next = null), (E1.prev = E0), (E0.prev = null), (E0.node = N0), (E1.node = N1),
(N0.inDeg = 0), (N1.inDeg = 2), (N0.outDeg = 1), (N1.outDeg = 1), (N0.inNodesList = null),
(N1.inNodesList = L0), (N0.outNodesList = L1), (N1.outNodesList = L1),
(N0.visitedInsNum = 2), (N1.visitedInsNum = 1)]
  boolean isAcyclic = true;                                        [.., isAcyclic = true]
  init();                                  [.., N0.visitedInsNum=0, N1.visitedInsNum= 0]
  ListEntry cur = nodes.head;                                             [.., cur = E0]
  if (cur != null) {
    ListEntry source=findSource(cur);                                  [.., source = E0]
    if (source == null) {
      isAcyclic = false;
    } else {
      fixIns(source.node);                 [.., N0.visitedInsNum=0, N1.visitedInsNum= 2]
      Node tmp = source.node;                                             [.., tmp = N0]
      source.node = cur.node;                                         [.., E0.node = N0]
      cur.node = tmp;                                                 [.., E0.node = N0]
      cur = cur.next;                                                     [.., cur = E1]
      if (cur != null) {
        source = findSource(cur);                                      [.., source = E1]
        if (source == null) {
          isAcyclic = false;
        } else {
          fixIns(source.node);             [.., N0.visitedInsNum=0, N1.visitedInsNum= 1]
          tmp = source.node;                                              [.., tmp = N1]
          source.node = cur.node;                                     [.., E1.node = N1]
          cur.node = tmp;                                             [.., E1.node = N1]
          cur = cur.next;                                               [.., cur = null]
          assume (cur == null)
        }
      }
    }
  }
  return isAcyclic;                               [.., topologicalSort$return = true]
[(G0.nodes = L0), (L0.head = E0), (L1.head = E1), (E0.next = E1), (E1.next = null)
(E1.prev = E0), (E0.prev = null), (E0.node = N0), (E1.node = N1), (N0.inDeg = 0)
(N1.inDeg = 2), (N0.outDeg = 1), (N1.outDeg = 1), (N0.inNodesList = null),
(N1.inNodesList = L0), (N0.outNodesList = L1), (N1.outNodesList = L1),
(N0.visitedInsNum = 0), (N1.visitedInsNum = 1)]
}
```

Figure 6-4: Program execution corresponding to the counterexample.

the program states. To simplify the representation, instead of giving the complete program states, Figure 6-4 only gives the state updates after each statement. The initial state and the post-states of the approximate call sites are the values given by the counterexample. Other program states are computed by symbolically executing each statement on its pre-state.

As shown at the end of the execution of Figure 6-4, the final state of the execution is the same as its initial state except for the values of the `visitedInsNum` fields of `N0` and `N1`. The final state violates the property because `int(E1.node'.inDeg)` equals 2 while `size(E1.*prev')` equals 1, and thus `int(E1.node'.inDeg)<size(E1.*prev')` does not hold. This execution, however, is valid only in the abstraction of the `topologicalSort` method; it is not valid in its original code because of the invalid call to the `fixIns` method in Line 9. The following section describes how the behavior of a call site is checked for validity.

6.1.2 Checking Call Sites

In order to check the validity of a counterexample, the original code of each procedure called in the program execution corresponding to that counterexample has to be checked against the state transition that the counterexample assigns to that call. Since the abstraction phase is based on the procedure call hierarchy of the code, the check for validity is performed hierarchically. That is, when a call to a procedure p is checked, all the procedures called by p are abstracted. They are checked for validity only after checking the call to p passes. We check the call sites of a counterexample in the depth first order (pre-order), i.e. the order in which they are actually called in the corresponding program execution.

A call to a procedure p is checked against a state transition (S, S') using Forge and Kodkod. The body of p is translated by Forge, encoding its call sites by their initial specifications, using the technique of Chapter 5. The generated formula is conjoined with an additional constraint that constrains the formal parameters of p to be equal to its actual arguments at the call site. The resulting formula is then checked for correctness by Kodkod. The values stored in the pre-state S and the post-state S' are encoded as a partial solution in Kodkod. Therefore, any solution found by Kodkod represents an execution of p that satisfies this state transition.

If Kodkod finds a solution, the validity of that solution has to be checked with respect to the internal call sites of p. If all internal call sites of p pass the check, the call to p has been validated. That is, (S, S') is a valid state transition for p. However, if checking any of the internal calls fails, that call will be marked as invalid and its specification will be refined (by the technique described in the next section). The procedure p will then be checked again against the state transition (S, S'). This process continues until either the call to p is validated, or it is shown to be inconsistent with (S, S'). In the latter case, the specification of p will be refined and the counterexample will be eliminated.

Example. We illustrate the above process by checking the validity of the counterexample of Figure 6-4. As mentioned before, that counterexample is invalid because of its invalid call to the `fixIns` method in Line 9. Here, we show how the invalidity of that call site is established[1].

We use Forge to translate the original code of the `fixIns` method (after two loop unrollings) to an Alloy formula. The code and its translation are given in Figures 6-5(a) and Figure 6-5(b). The last two lines of the formula of Figure 6-5(b) specify the actual arguments used at the call site being checked. Figure 6-5(c) gives the pre-state and the post-state that the counterexample assigns to that call site. These values are encoded as a partial solution in Kodkod.

We use Kodkod to solve the formula of Figure 6-5(b) with respect to the values of Figure 6-5(c). Kodkod cannot find any satisfying solution, implying that the pre and post states assigned to the call site are not consistent with the code of the `fixIns` method. Thus, this call site is marked as invalid; and its specification will be refined as explained in the next section.

[1] The call sites are checked in the order in which they are called. In this counterexample, the first call site (the call to `init`) does not need to be checked because its specification is accurate. The second call site (the call to `findSource`) has to be checked, but because it turns out to be valid, we illustrate checking the call to `fixIns` which is invalid.

```
void Graph.fixIns(Node n) {
1:  ListEntry p = n.outNodesList.head;
2:  if (p != null) {
3:      p.node.visitedInsNum = p.node.visitedInsNum + 1;
4:      p = p.next;
5:      if (p != null) {
6:          p.node.visitedInsNum = p.node.visitedInsNum + 1;
7:          p = p.next;
8:          assume (p == null);
9:      }
10: }
}
```

(a)

```
1: let $p_1$ = n.outNodesList.head,
3:   visitedInsNum$_1$ = visitedInsNum ++ $p_1$.node -> (int($p_1$.node.visitedInsNum) + 1),
4:   $p_2$ = $p_1$.next,
6:   visitedInsNum$_2$ = visitedInsNum$_1$ ++ $p_2$.node -> (int($p_2$.node.visitedInsNum$_1$) + 1),
7:   $p_3$ = $p_2$.next,
9:   visitedInsNum$_3$ = if ($p_2$ != null_rel) then visitedInsNum$_2$ else visitedInsNum$_1$,
      $p_4$ = if ($p_2$ != null_rel) then $p_3$ else $p_2$,
10:  visitedInsNum' = if ($p_1$ != null_rel) then visitedInsNum$_3$ else visitedInsNum,
      p' = if ($p_1$ != null_rel) then $p_4$ else $p_1$ in {
8:  ($p_1$ != null_rel)  =>  ($p_2$ != null_rel)  =>  ($p_3$ = null_rel)

binding actual values:
   n = source$1.node
   visitedInsNum' = visitedInsNum$1 }
```

(b)

```
pre-state:
[(L0.head = E0) (L1.head = E1) (E0.next = E1) (E1.next = null)
(E0.node = N0) (E1.node = N1) (N0.outNodesList = L1) (N1.outNodesList = L1)
(N0.visitedInsNum = 0) (N1.visitedInsNum = 0) (source$1 = E0)]

post-state:
[(L0.head = E0) (L1.head = E1) (E0.next = E1) (E1.next = null)
(E0.node = N0) (E1.node = N1) (N0.outNodesList = L1) (N1.outNodesList = L1)
(N0.visitedInsNum$1 = 0) (N1.visitedInsNum$1 = 2)]
```

(c)

Figure 6-5: The `fixIns` method: (a) code after two loop unrollings, (b) Alloy encoding and call site constraints, (c) pre- and post-state values

6.2 Specification Refinement

If the state transition that a counterexample assigns to a call to a procedure p is invalid, the specification of that call will be refined. The refined specification is a conjunction of p's old specification and a new specification that encodes the final values of some variables (or fields) in some execution paths of p, using relational expressions. That is, the new specification of p specifies the effects of some execution paths of p that were over-approximated by its old specification. As a result, the refined specification eliminates any counterexample that invokes any of those paths and assigns an invalid post-state to it.

The refinement technique that we introduce in this section guarantees that our instantiation of the specification refinement framework always terminates. This is because the refinement phase monotonically extends the specification of some procedure. The specification is an abstraction of the code, so in the limit, it will be equivalent to the code. Because the code and its heap are finitized, the limit will be reached in a finite number of steps. Since it might be the case that every refinement infers the value of only one variable (or field) in only one execution of the analyzed procedure, the number of refinements needed for the analysis can be exponential in the size of the code. Our experiments (discussed in the next chapter), however, suggest that the cost of analysis is small in practice.

Let a counterexample assign a state transition (S, S') to a call to a procedure p. If the translation of the body of p, represented by a formula f, does not have a solution satisfying (S, S'), f can be used as the specification of p to eliminate this state transition, and thus this counterexample. Using f as the specification of p, however, is in general equivalent to inlining the procedure p at the call site, and is not practical. Therefore, instead of using f, we use a formula weaker than f, namely an unsatisfiability proof of f (with respect to (S, S')), as the specification of p.

If an Alloy formula f is unsatisfiable with respect to a partial solution, Kodkod can generate a proof of unsatisfiability which is an Alloy formula weaker than f but still unsatisfiable with respect to the partial solution. Although the proof generated by

Kodkod is usually smaller than the original solved formula, it is still too conservative to use: it encodes the effects of several execution paths of the analyzed procedure that are not relevant to the state transition that is being eliminated.

Using the technique described below, we reduce the proof of unsatisfiability generated by Kodkod to a smaller Alloy formula whenever possible, and use this smaller formula as the new specification of the analyzed call site. This specification rules out the invalid state transition (S, S') by specifying how the procedure actually behaves on the given states. Therefore, it rules out not only this particular counterexample, but also all other counterexamples that invoke the same behavior of the procedure. The new specification is conjoined with the previous specification of the call site to form a refined specification.

In the rest of this section, we describe how a proof of unsatisfiability generated by Kodkod is reduced to a smaller well-formed Alloy formula. The technique is illustrated by an example.

6.2.1 Generating Small Unsatisfiability Proofs

Given an unsatisfiable formula, Kodkod can return a proof of unsatisfiability as a well-formed Alloy formula. However, due to the nature of the Alloy formulas that we analyze (containing complex nested if expressions), the proof generated by the current implementation of Kodkod is too conservative to use; it can often be reduced substantially. Therefore, instead of directly using the proof returned by Kodkod, we use it to extract some information and construct a smaller proof, also as a well-formed Alloy formula.

We use the proof generated by Kodkod to compute a set *Irrelevant*, consisting of the subformulas and subexpressions of the analyzed formula that are not relevant to the unsatisfiability of that formula. We then use *Irrelevant* to construct a small proof.

A proof returned by Kodkod might have been generated by two sources: (1) Kodkod's internal reductions, and (2) a proof generated by a SAT solver. In each case, the set *Irrelevant* can be computed as follows.

- **Internal reductions.** As described in Section 5.4, Kodkod translates an Alloy formula to boolean by allocating boolean variables for the tuples that may be contained in each relation. However, instead of boolean variables, it uses the boolean constants *true* and *false* for those tuples that are in the lower bound and those not in the upper bound of a relation. During the translation of a formula, these constant values are propagated through matrix operations to simplify the resulting boolean expressions. As a result of these propagations, it is possible for an Alloy formula to be simplified to either *true* or *false* at the top level. In this case, Kodkod returns the result immediately without calling a SAT solver.

  If solving an Alloy formula with respect to a partial solution simplifies to *false*, the formula is unsatisfiable. In this case, Kodkod returns an unsatisfiable subformula of the solved formula as a proof of unsatisfiability. We use this proof to compute the set *Irrelevant*: we evaluate every subexpression and subformula of the returned proof with respect to the partial solution; any subexpression and subformula that evaluates to a constant value that causes the the top-level formula to simplify to *false* is involved in the unsatisfiability. Otherwise, it is not involved and thus will be included in *Irrelevant.*

- **SAT solver's proof.** If the analyzed Alloy formula does not simplify to a constant, it is handed to a SAT solver for a solution. Given an unsatisfiable formula, some SAT solvers can generate a proof of unsatisfiability consisting of boolean clauses. For this purpose, we configure Kodkod to use the ZChaff SAT solver [45] as its backend engine. Given an unsatisfiable boolean formula in the CNF format, ZChaff is capable of generating a proof of unsatisfiability called an *unsat core* [63]. The unsat core is a subset of the clauses in the solved CNF that is unsatisfiable and therefore, is a witness to the unsatisfiability of the original formula. Although the core generated by ZChaff is not necessarily minimal, it is usually much smaller than the original formula that was solved.

  Given an unsat core, we compute the set *Irrelevant* as follows: an Alloy subexpression or subformula x is involved in the unsatisfiability of the analyzed formula if any of the boolean variables encoding x is included in the core. Furthermore, if x simplifies to a constant value under the given partial solution, it is considered to be involved in the unsatisfiability of the analyzed formula. If neither of these cases holds, x will be included in *Irrelevant*.

Given an unsatisfiable Alloy formula f and a set *Irrelevant* of the subexpressions and subformulas that are not involved in the unsatisfiability of f, we reduce f to a proof of unsatisfiability by replacing any subformula of f included in *Irrelevant* with an unconstrained boolean variable[2], and any subexpression of f included in *Irrelevant* with an unconstrained relation of the appropriate type. Thus, the resulting formula constrains only the parts that have actually caused the unsatisfiability.

This reduction, however, generates many new relations. To reduce the number of new relations, we perform the following optimizations whenever possible: (1) if a subformula s included in *Irrelevant* can be removed from f without making it malformed (e.g. if it is part of a top-level conjunct), we remove s without introducing any boolean variable to replace it, and (2) if a subexpression s included in *Irrelevant* is the right-hand side of an equality, we replace s with the relation of the left-hand side, thus allowing that relation to take any arbitrary value. No new relation is introduced in this case.

To clarify how we replace subexpressions and subformulas, consider the following formula:

$$\texttt{v = if (}cond_1\texttt{) then } e_1 \texttt{ else if (}cond_2\texttt{) then } e_2 \texttt{ else } e_3$$

where $cond_1$ and $cond_2$ represent two formulas, and e_1, e_2, and e_3 represent some relational expressions. Suppose that this formula is unsatisfiable with respect to a given partial solution, and thus we have to compute a proof of unsatisfiability for it. Let us assume that the set *Irrelevant* contains the following elements:

$$Irrelevant = \{cond_2, e_1\}$$

[2] More precisely, a singleton unary Alloy relation of type boolean.

Given this set, the analyzed formula is reduced to the following proof of unsatisfiability in which choice is a fresh boolean variable representing a non-deterministic choice and r is a fresh relation of the same type as e_1.

v = if ($cond_1$) then r else if (choice) then e_2 else e_3

This is a well-formed Alloy formula that leaves the irrelevant parts unconstrained. However, the above-mentioned optimization rules allow us to avoid introducing r and reuse v instead. Thus the generated proof of unsatisfiability will be the following:

v = if ($cond_1$) then v else if (choice) then e_2 else e_3

That is, if $cond_1$ holds, any arbitrary value of v is accepted. Otherwise, either e_2 or e_3 is chosen nondeterministically for the value of v.

6.2.2 Example

As mentioned before, the counterexample of Figure 6-4 is invalid because its first call to the fixIns method is invalid. As described in the previous section, this call site was marked as invalid because the the formula of Figure 6-5(b) was unsatisfiable with respect to the pre- and post-state values of Figure 6-5(c). For this example, Kodkod generates the following proof of unsatisfiability:

(source$1 = E0) (E0.node = N0) (E1.node = N1) (N0.outNodesList = L1)
(L1.head = E1) (E1.next = null) (N1.visitedInsNum = 0) (N1.visitedInsNum$1 = 2)

let p_1 = n.outNodesList.head,

  visitedInsNum$_1$ = visitedInsNum ++ p_1.node -> (int(p_1.node.visitedInsNum) + 1),

  p_2 = p_1.next,

  visitedInsNum$_2$ = visitedInsNum$_1$ ++ p_2.node -> (int(p_2.node.visitedInsNum$_1$) + 1)

in {

  visitedInsNum$1 = **if** ($p_1$!= null_rel) **then if** (p_2 != null_rel) **then** visitedInsNum$_2$

      else visitedInsNum$_1$ **else** visitedInsNum

  n = source$1.node }

The first two lines specify those components of the pre- and post-state that cause the unsatisfiability[3]. The other lines give two constraints, one specifying the final value of visitedInsNum and the other constraining the formal parameter n to be equal to the actual parameter source.node. These constraints evaluate to *false* because the value of N1.visitedInsNum$1 given by the translation constraint conflicts with the one given in the post-state; the former simplifies to the integer 1 whereas the latter is the integer 2.

We reduce this proof to a smaller proof using the technique described before. In this example, the unsatisfiability proof is a result of Kodkod's internal reductions. Because the first branching condition (p_1 != null$rel) simplifies to *true* and the second condition (p_2 != null$rel) simplifies to *false*, the else branch of the first conditional and the then branch of the second conditional are not involved in the unsatisfiability of the formula. That is,

$$\mathit{Irrelevant} = \{\ \texttt{visitedInsNum}_2, \texttt{visitedInsNum}\ \}$$

Therefore, the proof of unsatisfiability is reduced to the following formula:

```
let p1 = n.outNodesList.head,
    visitedInsNum1 = visitedInsNum ++ p1.node -> (int(p1.node.visitedInsNum) + 1),
    p2 = p1.next
in {
    visitedInsNum$1 = if (p1 != null_rel) then if (p2 != null_rel) then visitedInsNum$1
                      else visitedInsNum1 else visitedInsNum$1
    n = source$1.node }
```

That is, the value of visitedInsNum$1 is not constrained in the branches that are irrelevant to the unsatisfiability of the formula.

This unsatisfiability proof is the new specification of the fixIns method. It is conjoined with its previous specification, and used to check the topologicalSort method against the property. In this example, no further counterexamples are found.

[3] The proof returned by Kodkod does not actually include these values. However, we include them as parts of the proof to make the proof easier to understand.

Therefore, the property is validated and the analysis terminates. No further specification refinements are needed.

Chapter 7

Experiments

We have implemented our specification refinement technique in a prototype tool called Karun, and used it to evaluate our technique from different aspects. Two case studies have been done to show that the specification refinement technique is applicable to real Java APIs, and a set of comparisons has been performed to assess its performance. Furthermore, some experiments have been performed to evaluate the accuracy and effectiveness of the abstract interpretation technique that we use to extract initial procedure summaries. This chapter summarizes the results of these experiments.

7.1 Overview

To evaluate our analysis technique, we have developed a prototype tool called Karun. It takes a Java program in which a procedure is selected for analysis, a property expressed in Alloy that specifies a pre- and post-condition for the selected procedure, and two types of bounds: one on the number of iterations of the loops in the code, and the other on the number of objects considered of each type.

Karun implements the techniques described in Chapters 4 to 6. That is, it uses the abstract interpretation technique of Chapter 4 to extract initial specifications of the procedures that are called directly or indirectly by the procedure being analyzed. It then uses Forge, as described in Chapter 5, to construct an Alloy formula that abstracts the analyzed procedure, and Kodkod to check the resulting formula against the property of interest. As described in Chapter 6, Karun checks the validity of any found counterexample using Kodkod again, and refines procedure specifications based on the proofs of unsatisfiability that are generated by the SAT solver used as Kodkod's engine. Although we use the ZChaff SAT solver in all of our experiments, any SAT solver capable of generating proofs of unsatisfiability can potentially be used by Karun.

We evaluated our analysis technique with different sets of experiments. The first set of experiments used Karun to check some properties of two open source Java APIs: a job scheduler, Quartz [2], and a graph package, OpenJGraph [1]. Typical of Java code, in both APIs, the procedures make many calls – in one case, a top-level procedure invoked 81 other procedures directly or indirectly – and use a variety of data structures from the Java standard library.

We checked the procedures of the Quartz API against properties that we extracted from the informal comments available in the code. Our experiments uncovered two previously unknown bugs in this API that are actually observable by Quartz users. Both bugs involve deeply structural properties of the code, and are caused by the errors in some called procedure rather than the top-level procedure analyzed. We reported the bugs to the Quartz developers and they fixed them.

The second Java API that we analyzed, namely OpenJGraph, implements some well known graph algorithms. Therefore, we were able to extract the correctness properties of the API procedures from a widely used algorithms textbook [12]. In addition to checking these correctness properties, we also checked the code with respect to some representation invariants extracted from the informal comments available in the code. All checks passed successfully with respect to the analyzed bounds.

In a second set of experiments, we evaluated the performance of our analysis technique. We compared Karun with our implementation of another technique that also translates the code to Alloy and checks it for a counterexample using Kodkod. This technique, however, does not abstract procedure calls; it inlines all call sites reachable from the analyzed procedure. Therefore, the results of this comparison show how the on-demand specification inference technique affects the performance of the analysis. Our experiments suggest that our technique substantially reduces the analysis time when the analyzed property is partial. However, in checking full properties of the code, a simple analysis that inlines all procedures is more effective.

We performed a third set of experiments to evaluate our abstract interpretation technique that extracts initial procedure summaries. These experiments evaluate both the accuracy and the effectiveness of the extracted summaries. The accuracy is evaluated by checking if the summaries generated for a number of procedures are sufficient to prove their specifications. For this purpose, we used the standard Java implementation of linked list (whose procedures are small, but structurally complex), and the OpenJGraph API (whose procedures use different data structures and make many calls). In 17 of the 30 procedures, the generated summaries were sufficient to check the full specifications; in 16 procedures, the generated summaries included all frame conditions, and provided some partial specifications. Only in one case there was a significant loss of information.

We evaluated the effectiveness of our abstract interpretation technique by comparing the extracted summaries against *frame conditions* – specifications that only constrain which variables and fields are not modified by a procedure. We verified a number of procedures using our specification refinement technique, once starting from the summaries extracted by abstract interpretation, and once starting from frame conditions. Our experiments show that in almost all cases the summaries extracted by abstract interpretation result in a substantially reduced analysis time. This implies that not only are they effective in ruling out many invalid executions of the abstract code, but also that the overhead of extracting these summaries is negligible in the overall analysis time. These results suggest that our abstract interpretation technique represents a useful balance between tractability and accuracy.

7.2 Case Studies

In this section, we describe the experiments that we performed on two open source Java APIs, namely Quartz and OpenJGraph. Typical of Java code, they both use a

```
class java.util.Set {

 /*
  * contains$rel: Set --> Data
  */

 /* contains$rel' = contains$rel + (this --> d)
  * add$return = if (d in this.contains$rel) then false$rel else true$rel */
  boolean add(Data d) {..}

 /* contains$rel' = contains$rel - (this --> d)
  * remove$return = if (d in this.contains$rel) then true$rel else false$rel */
  boolean remove(Data d) {..}

 /* contains$return = if (d in this.contains$rel) then true$rel else false$rel */
  boolean contains(Data d) {..}

 /* contains$rel' = contains$rel - (this --> univ) */
  void clear() {..}

 /* size$return = size(this.contains$rel) */
  int size() {..}

}
```

Figure 7-1: Specifications used for the `Set` data structure.

variety of data structures from the Java standard library. Although we could have treated the calls to the library like other method calls of the code (that is, to infer their specifications on-demand using their actual implementations), we chose to provide their specifications manually instead. This allowed the analysis to be performed over abstract representations of the data structures, rather than their actual implementations and thus made the analysis more efficient and properties easier to write. However, replacing library calls with their specifications depends on the assumption that the implementations provided by the library actually conform to the specifications used. Therefore, a separate analysis (e.g. similar to the one performed by Dennis, et. al. [14]) is needed to check the library implementations against their specifications.

Figures 7-1 and 7-2 show the specifications that we used for the `Java.util.Set` and `Java.util.Map` data structures, respectively. Figure 7-1 gives the specifications of a `Set` data structure that stores objects of type `Data`. It uses a binary relation contains$rel to represent the contents of sets. Unlike the relations declared for the fields of user-defined datatypes, this relation is not constrained to be functional. That is, it can map a Set object to any arbitrary number of Data objects, thus allowing sets of cardinality more than one.

For each operation defined for the Set datatype, we have to provide a specification. Figure 7-1 shows the ones that are widely used. The specification of the add operation ensures that in the post-state, the receiver set contains the argument data. It returns true if the set did not already contain that data, and false otherwise. Similarly the remove operation is specified by ensuring that the given data is not included in the receiver set in the post-state. It returns true if the set contained the given data, and false otherwise. The specification of the contains method checks for set membership using an Alloy if expression. The clear method is specified by removing all data from the receiver set. Finally, the return value of the size method is specified by the set cardinality operator in Alloy.

Figure 7-2 gives the specifications that we use for the java.util.Map data structure, mapping objects of type Key to those of type Data. It uses a ternary relation mapping$rel to represent the contents of maps. This relation maps each object of type Map to an arbitrary number of pairs of type Key -> Data. The specifications of get, put, remove, containsKey, containsValue, clear and size are straightforward: the updates to a map are specified by adding and removing tuples to and from the relation mapping$rel, and memberships are tested by forming Alloy if expressions. The specifications of keySet and values, however, depend on our representation of Set. They return a fresh object of type Set, denoted by New$Set, whose contents are constrained to be the keys (given by this.mapping$rel.univ) or the values (given by univ.(this.mapping$rel)) stored in the receiver map.

As shown in Figures 7-1 and 7-2, the types of the specification relations that abstract data structures of the Java library depend on the types of the contents of those data structures. In the Java library, however, these data structures are generic, and the actual types of their contents are declared only in the code using them. Since this type information is lost in the bytecode, we have to get that information from

```
class java.util.Map {

  /*
   * mapping_rel : Map --> Key --> Data
   */

  /* get$return = if some k.(this.mapping$rel) then k.(this.mapping$rel) else null$rel */
   Data get(Key k) {..}

  /* mapping$rel' = mapping$rel - (this --> k --> univ) + (this --> k --> d)
   * put$return = if some k.(this.mapping_rel) then k.(this.mapping$rel) else null$rel */
   Data put(Key k, Data d) {..}

  /* mapping$rel' = mapping$rel - ((this --> k) --> univ)
   * remove$return = if some k.(this.mapping$rel) then k.(this.mapping$rel) else null$rel */
   Data remove(Key k) {..}

  /* containsKey$return = if some k.(this.mapping$rel) then true$rel else false$rel */
   boolean containsKey(Key k) {..}

  /* containsValue$return = if (some this.mapping$rel.d) then true$rel else false$rel */
   boolean containsValue(Data d) {..}

  /* mapping$rel' = mapping$rel - (this --> univ --> univ) */
   void clear() {..}

  /* size$return = size(this.mapping$rel) */
   int size() {..}

  /* keySet$return = New$Set
   * contains$rel' = contains$rel + (New$Set --> (this.mapping$rel.univ)) */
   Set keySet() {..}

  /* values$return = New$Set
   * contains$rel' = contains$rel + (New$Set --> univ.(this.mapping$rel)) */
   Set values() {..}

}
```

Figure 7-2: Specifications used for the `Map` data structure.

the source code prior to the analysis. That is, before analyzing a piece of Java code, we syntactically modify its source code so that instead of using the generic datatypes of the Java library, it uses our library specifications with appropriate contents types. It should be noted that it is possible to take an alternative approach in which the specification relations are also declared to be generic, i.e. they are declared over the type `Object`. However, the `Object` type is the union of all other types used in the analyzed code and can get very big. Therefore, solving an Alloy formula that contains relations over this type can become prohibitively costly.

7.2.1 Quartz API

Quartz [2] is an open source Java API apparently used by thousands of people. It provides a job scheduling facility that can be integrated with different applications to repeatedly execute a set of jobs according to their schedules. For example, a backup job can be defined that saves the contents of a database to a backup storage at 7pm on all weekdays except holidays.

We have checked 4 classes of this package that provide some core functionalities of Quartz. The analyzed classes are as follows:

- The `QuartzScheduler` class is the main class responsible for scheduling jobs, executing them, and registering and unregistering their listeners.

- The `SimpleTrigger` and `CronTrigger` classes provide two different types of triggers. Triggers are used to schedule jobs: they specify at what times and how many times a job should be fired (executed). A job can have multiple triggers, but each trigger has exactly one job associated with it. `SimpleTrigger` provides a simple schedule that can be represented by the start time, end time, repeating intervals, and the number of repeats. `CronTrigger` provides a more complex schedule represented by seconds, minutes, hours, day of the week, day of the month, month, and year. Furthermore, a trigger may be associated with a calendar that disallows firing jobs at certain times.

```
/**
* Get the names of all of the jobs that have the given group name.
*/
public List<String> RAMJobStore.getJobNames(SchedulingContext ctxt,
                        String groupName) {
   ..
}
```

(a)

```
getJobNames$return.contents' =
univ.(groupName.(this.jobsByGroup.mapping$rel).mapping$rel).jobDetail.name
```

(b)

Figure 7-3: An example of (a) an informal comment, and (b) its corresponding formalization in Alloy.

- The `RAMJobStore` class stores jobs and their triggers in the RAM, allowing a fast access to jobs. The data, however, is not persistent between program shut-downs.

We have checked a total of 40 public methods in these classes, requiring 3900 lines of source code to compile. 15 public methods, however, cannot be handled by the current implementation of Karun. This is mainly because of Karun's lack of support for recursion, string manipulating operations, and arithmetic expressions other than integer additions and subtractions.

We checked each method against a property extracted from the informal comments available in the code. That is, we formalize the comments as formulas in Alloy and use them to check the correctness of the methods. Whenever possible, the property is broken into several partial properties that are then checked separately.

Figure 7-3 provides an example of an informal comment and its corresponding Alloy formula. The comment states that the method is supposed to return the names of all jobs with a specific group name. The formula formalizes this comment in terms of the contents of the two `Map` data structures that are involved in the code. This example shows that the formal properties are expressed in terms of the fields of the data structures that are actually used in the implementation.

procedure	property	heap size/ unrollings	time (sec)
getTriggerNames	returns the triggers with the group name	5/3	23
getJobNames	returns the jobs having the group name	5/4	87
removeCalendar	if no trigger has the calendar, removes it	5/5	95
getTriggersOfJob	returns triggers associated with a job	4/4	77
resumeJob	resumes only jobs with given name/group	3/3	55
scheduleJob	returns a date in calendar of given trigger	3/3	11
storeCalendar	recomputed firing times belong to calendar	4/2	65

Table 7.1: A sampling of the properties analyzed in Quartz, their bounds, and analysis time.

We checked every property of a method with respect to different bounds for loop iterations and the heap size. In most cases, increasing the bounds increases the analysis time exponentially. In all of our experiments we increase the bounds until the analysis time exceeds 15 minutes (using a Pentium 4, 1.8GHz with 512 MB of RAM). Table 7.1 gives a sampling of the properties, bounds, and the analysis times. The bounds give the maximum number of objects of each type and the maximum number of loop iterations considered in the analysis. The time column gives the total analysis time in seconds.

Our experiments uncovered two previously unknown bugs in the Quartz API that are actually observable by Quartz users. Both bugs involve deeply structural properties of the code, and are caused by the errors in some called method rather than the top-level method analyzed.

The first bug was found while checking the `storeJob` method shown in Figure 7-4. The method takes a job argument (`newJob`) and stores it in the job store implemented by a map data structure (`jobsByFQN`) if the job does not already exist in the store or if the boolean argument `replaceExisting` is set to true. The corresponding Alloy property is shown in that figure too. It specifies that if either of the above conditions holds, after the execution of the method, the store should contain a job which is equal to the argument job. The subformula `preCondition` mentioned in this property represents some representation invariants that ensure different data structures used by the code are initially consistent. Checking this property with respect to 7 objects

```
/**
* Store the given job.
*
* @param newJob: the job to be stored.
* @param replaceExisting: if true, any job existing in the jobStore with the
*             same name and group should be over-written.
*/

/* @property:
 * preCondition =>
 * (((key(newJob) !in this.jobsByFQN.mapping$rel.univ) or
 *   (replaceExisting = true)) =>
 * (some jw: JobWrapper | jw in univ.(this.jobsByFQN'.mapping$rel') and
 *                        jobDetailsEqual(newJob, jw.jobDetail')))
 */

public void storeJob(JobDetail newJob, boolean replaceExisting, ..) {
    JobWrapper jw = new JobWrapper(newJob.clone());
    ..
    jobsByFQN.put(jw.key, jw);
}
```

Figure 7-4: The first buggy method found.

of each type and one loop unrolling results in a counterexample.

Examining the counterexample shows that the property does not hold because the order of the job listeners associated with the stored job may be different from that of the argument job, and thus the two jobs are not exactly equal. This wrong behavior is a result of a bug in the `clone` method which is called by `storeJob`. The `clone` method does not necessarily preserve the order of job listeners while cloning a job's information. This bug is actually observable by the users of Quartz. We reported the bug, and the developers fixed the code. Checking the property in the fixed method does not produce any more counterexamples.

The second bug was found when the `storeCalandar` method shown in Figure 7-5 was checked. The loop is simplified to improve readability. The method takes a calendar and stores it in a calendar store. In addition to that, if the `updateTriggers` argument is set to true, it recomputes the next firing time of all triggers associated with the given calendar name so that the new firing time belongs to the new calendar. The Alloy property given in Figure 7-5 is not the full specification of the code; it is

```
/**
* Store the given calendar.
*
* @param calendar: the calendar to be stored.
* @param updateTriggers: if true, all triggers existing in the jobStore that reference
*                        an existing calendar with the same name will have their
*                        next fire time re-computed with the new calendar.
*/

/* @property:
 * preCondition =>
 * (all t: Trigger | (t.nextFireTime' != t.nextFireTime) =>
 *     ((t.nextFireTime' = null) || included(t.nextFireTime', calendar)))
 */

public void storeCalendar(String name, Calendar calendar,
                                    boolean updateTriggers, ..) {
    ..
    for (Trigger trig : getTriggersForCalendar(name)) {
        ..
        trig.updateWithNewCalendar(calendar, getMisfireThreshold());
    }
    ..
}
```

Figure 7-5: The second buggy method found.

only a partial specification of how firing times are updated. It specifies that any firing time recomputed by this method should either be set to `null` (meaning that it will never be fired again) or belong to the new calendar. Checking the code against this property with respect to 4 objects of each type and 1 loop unrolling results in a counterexample, witnessing a case where the firing time is updated but is not included in the given calendar.

Examining the counterexample shows that this wrong behavior is a result of a bug in the `updateWithNewCalendar` method implemented in both `CronTrigger` and `SimpleTrigger` classes, two subclasses of `Trigger`. In fact, the two implementations are copies of each other. This method may be called by the dynamic dispatch of the call site in `storeCalendar`. This bug is also observable by Quartz users. We reported the bug and the developers fixed the code in both classes. Checking the fixed code did not generate any more counterexamples.

7.2.2 OpenJGraph API

We performed a second case study using the OpenJGraph package [1]. This package is an open source Java API that provides some popular algorithms (e.g. graph traversals, shortest paths, and minimum spanning trees) for different kinds of graphs (e.g. directed, undirected, and weighted). We have checked all the methods in the following two classes of this API:

- The `GraphImpl` class provides the core implementation of the graph data structure. It represents a graph as a map from each vertex to its list of adjacent edges and uses a separate set to record all the edges of the graph.

- The `MinimumSpanningTreeKruskalAlgorithm` class implements Kruskal's algorithm for computing a minimum spanning tree of a graph.

We checked a total of 20 methods in these two classes, requiring 2200 lines of source code to compile. All the analyzed methods are publically available to the API users. Most of them make several calls. The minimum spanning tree method, for example, invokes 81 other methods directly or indirectly.

The methods of the `GraphImpl` class were checked against the properties extracted from the informal comments available in the code. That is, we formalized the comments as Alloy formulas manually and checked if the methods satisfied those properties. The minimum spanning tree method, however, was checked against a set of partial properties extracted from a widely used algorithms textbook [12].

Table 7.2 gives a sampling of the analyzed properties along with the bounds, and the analysis time. We checked each property with respect to several combinations of the bounds on the number of objects of each type and the number of loop iterations. We increased the bounds until the analysis time exceeded 15 minutes.

In addition to the correctness properties, we also checked each method against a set of representation invariants: the consistency conditions that should hold among the data structures used in the code. The representation invariants are extracted from the informal comments available in the code and are checked by properties

procedure	property	heap size/ unrollings	time (sec)
add(v)	edge list of v is either empty or unchanged	4/4	71
addEdge(e)	if e added, e's both ends belong to graph	4/4	19
addEdge(v1, v2)	edge lists of other vertices unchanged	4/4	46
removeEdge(e)	only edge lists of e's ends may change	4/4	67
removeEdge(e)	e is removed from edge lists of its ends	4/4	13
minSpanTree()	edges of returned mst are in original graph	5/5	35
minSpanTree()	vertices with no edges are not in mst	5/5	154

Table 7.2: A sampling of the properties checked in OpenJGraph.

that specify that if a representation invariant holds in the pre-state, it holds in the post-state too. Some of the analyzed representation invariants are listed below:

- The set of edges of a graph is the union of the edges in the adjacency lists of individual vertices.
- An undirected edge between two vertices v_1 and v_2 is contained in the adjacency lists of both v_1 and v_2.
- Each edge appears at most once in each adjacency list.
- An edge can be contained only in the adjacency lists of its two ends.

All methods analyzed in OpenJGraph successfully passed all the checks. That is, no counterexamples were found within the analyzed bounds.

7.2.3 Discussions

Our case studies show that Karun can be used to check complex correctness properties of the code even when the code accesses several data structures and makes many external calls. It is capable of finding errors not only in the top-level procedure being analyzed, but also in the called procedures if they prevent the top-level procedure from satisfying the property of interest.

Since Karun does not require users to provide any intermediate annotations, the only cost involved in using Karun is to specify the property. Our experiments suggest that the correctness properties of the programs manipulating data structures can be efficiently expressed in a relational logic like Alloy that provides a transitive closure operator. All properties (including the representation invariants) checked in our case studies were written as succinct Alloy formulas: they contained only 15 relational operators on average.

Although properties are often expressed as small Alloy formulas, they usually encode complex behaviors of the underlying code. Therefore, in order to write them, one should know exactly which data structures are involved in producing the behavior of interest and how those data structures are actually implemented (or how they are abstracted in case of library data structures). That is, it often requires deep knowledge about the code. In fact, it took the author of this thesis 4 weeks to understand the Quartz package well enough to write the analyzed properties. Therefore, the cost of applying our analysis technique will be minimized if it is used by programmers or those who have the required knowledge about the implementation details of the code.

7.3 Evaluating Performance

We performed a set of experiments to evaluate the improvements gained by our specification refinement approach. In these experiments, we compared the performance of Karun with that of a technique that is identical in all respects except that it inlines called procedures. In order to achieve a fair comparison, we shared code between the two implementations whenever possible. That is, they both use Alloy as an intermediate language, Kodkod as the constraint solver, and the ZChaff SAT solver as the backend engine.

We analyzed some procedures of the OpenJGraph and Quartz APIs using both implementations. The results of these experiments are given in Table 7.3. The `objs/unrolls` column indicates the maximum number of objects considered for each type and the number of times the loops are unrolled during the analysis. The time columns give the total analysis time (including all refinements in the specification refinement case) in seconds. The `refs` column shows the number of refinements needed

			inline	spec ref.		
procedure	property	objs/	time	time	refs	speed-
		unrolls	(sec)	(sec)		up
scheduleJob	returns valid date	3/3	>900	11	5	>82
addEdge	others unchanged	4/4	110	46	0	2.4
removeEdge	edge removed	4/4	271	13	4	20.8
minSpanTree	subgraph	5/5	>900	36	0	>25
resumeJobGrp	others not resumed	3/2	>900	115	49	>8
storeCalendar	triggers updated	4/2	>900	65	45	>14
remove Calendar	calendar removed if not in use	5/5	46	95	5	0.5
retrieveJob	correct job returned	7/1	64	166	19	0.4
getJobGrpNames	correct groups returned	7/7	621	>900	-	<0.7

Table 7.3: Experimental results: performance evaluation.

to analyze each procedure by the specification refinement technique. The last column summarizes the speedup gained as a ratio between total analysis times without and with specification refinement.

The first 6 rows of the table, where the speedup is greater than 1, give the results of checking *partial* properties of the procedures. That is, the properties involve only a subset of the procedures' code. Current experiments suggest that in those cases, the specification refinement technique always improves the analysis time substantially. In fact, as shown in this table, the initial summaries extracted by the abstract interpretation technique are sometimes sufficient to check a property; no further refinements are needed. Although in order to check each of these properties only a small portion of code need be analyzed, the inlining version spends considerable time to translate the whole code into a boolean formula. Consequently, the generated formula is much bigger than needed, and thus harder for the SAT solver to check.

Other rows of the table (the last 3 ones) give the results of analyzing *full* properties, those that depend on the complete code of all called procedures. In these cases, our specification refinement technique must perform several refinements to infer the full specifications of all call sites, and thus a simple analysis that inlines all called procedures upfront performs much better.

As shown in Table 7.3, checking different properties requires different number of refinements. In general, the number of required refinements depends on the amount of code that is related to the property being checked. Furthermore, it also depends on the order in which the SAT solver finds counterexamples: to rule out some invalid counterexamples, a rich specification is inferred that prevents some future counterexamples too.

These experiments suggest that when the analyzed property is partial, our specification refinement technique considerably improves the analysis timed by (1) reducing the translation time, and (2) generating smaller boolean formulas that can be checked faster.

7.4 Evaluating Initial Summaries

In this section, we describe the experiments that we performed to evaluate the quality of the initial procedure summaries that we generate. Summaries are evaluated from two aspects: accuracy and effectiveness. We evaluate the accuracy of the generated summaries by checking if they are sufficient to prove correctness properties of the code they summarize. The effectiveness is evaluated by comparing the generated summaries with frame conditions of the code.

The experiments were done using the following constants for the abstract interpretation parameters:

- Maximum number of operators allowed in an expression before it is widened to the universal relation is set to 1300.

- Maximum number of allocations enumerated before an uninterpreted set of objects is allocated is set to 5.

- Maximum number of unions before widening to closure is set to 3.

- Maximum number of contexts for which a procedure is summarized is set to 5.

7.4.1 Accuracy

We ran our abstract interpretation technique to generate summaries for a number of procedures from two Java libraries: the standard Java implementation of linked list and the OpenJGraph API. To evaluate the accuracy of the generated summaries, we used them to check properties of their corresponding procedures. We performed two experiments: (1) we summarized small, structurally complex procedures and compare those summaries to pre-existing full specifications. For this study, we used the Java linked list library. (2) we summarized procedures with more typical object-oriented code which uses several different data structures and makes many external calls. Since full specifications were not available, we assessed the summaries against manually written partial specifications. For this study, we used the OpenJGraph package. The accuracy of the generated summaries varied, but no summary took more than 3 seconds to generate even though some procedures had as many as 81 procedure calls.

The summaries were analyzed using Kodkod: for a procedure p and a specification S we checked if the formula $(\mathit{summary}(p) \implies S)$ holds, where $\mathit{summary}(p)$ is the summary generated by our abstract interpretation technique. Since Kodkod analyzes formulas with respect to finite scopes, we first checked the summaries in a high scope and then inspected them manually. Our technique generated full specifications for 13 of the 30 procedures. In the remaining 17 procedures, our technique generated all frame conditions, and in 16 of them it inferred some major properties. In one of the procedures the generated summary was too rough to be useful for more than its frame conditions.

Java Linked List Implementation. We generated summaries for 10 procedures in Sun's standard implementation of linked list in the Java Collections Framework. The implementation represents a list as a circular doubly linked list with a dummy header entry. An integer field `size` keeps track of the size of the list. Since our current technique does not handle arithmetic expressions (it widens all of them to the set of all possible integers `Int`), we ignored all accesses to `size`, and only analyzed

```
/*
 clone$return = New$LinkedList
 element' ⊇ element
 element' ⊆ element + (NewSet$Entry —> (this.header.^next & (Entry'- this.header)).element)
 next' ⊇ next
 next' ⊆ next + (NewSet$Entry —> NewSet$Entry)
 previous' ⊇ previous
 previous' ⊆ previous + (NewSet$Entry —> NewSet$Entry)
 header' = header ++ (New$LinkedList —> NewSet$Entry)
 Entry' ⊇ Entry
 Entry' ⊆ Entry + NewSet$Entry
 LinkedList' = LinkedList + New$LinkedList
*/

public Object clone() {
   LinkedList clone = new LinkedList();
   for (Entry e = header.next; e != header; e = e.next) {
      Entry newEntry = new Entry(e.element, clone.header, clone.header.previous);
      newEntry.previous.next = newEntry;
      newEntry.next.previous = newEntry;
   }
   return clone;
}
```

Figure 7-6: The `clone` method.

procedures that do not depend on integer arithmetic.

We generated summaries for 10 methods: `add`, `addFirst`, `addLast`, `remove`, `removeFirst`, `removeLast`, `getFirst`, `getLast`, `clear`, and `clone`. These methods are small and mostly self-contained, but their correctness depends on their complex mutations of the underlying doubly linked data structure.

For each method, we checked if the generated summary is accurate enough to show that the method's JML specification [23] holds. In 8 of the 10 analyzed methods, we were able to check the complete specifications. The summaries generated for `clone` and `remove` were not accurate enough to prove the full specifications, although they still provided limited descriptions. These methods cause our technique difficulty because they mutate the same relation that is being traversed in a loop.

The summary generated for the `clone` method, shown in Figure 7-6, specifies that a fresh list is constructed and returned, and that it contains some freshly allocated entries whose elements are chosen from the set of the elements of the receiver list.

Although the summary does not specify that the size of the constructed list is the same as the receiver list nor that the copy preserves order, it still specifies useful information about the method – for example, that the elements of the returned list are all chosen from the receiver list, and that the receiver list is not changed at all.

The `remove` method removes the first occurrence of a given element from the list. Although the list is updated at most once (for the first occurrence), the update is done as part of the loop. This prevents the analysis of the loop from stabilizing by inferring the closure, and the loop analysis instead terminates by widening to the universal relation. That is, the generated summary allows the values of the `next` and `previous` fields of any `Entry` object to change, meaning that any number of entries may be removed from the list. The summary therefore only provides the frame conditions.

OpenJGraph API. We used our abstract interpretation technique to generate summaries for 20 procedures of the OpenJGraph package. Instead of generating summaries for the Java map and set data structures using their implementations, we provided their specifications in the style of the summaries that our technique generates. Doing so demonstrates how our technique is compositional and can exploit off-the-shelf specifications when available.

Most procedures in this package make a considerable number of external calls, either directly or indirectly. Our technique generates full specifications for 5 of the 20 procedures. In order to evaluate the quality of the summaries generated for the other 15 procedures, we checked them against two sets of properties:

- *Representation invariants.* The graph package assumes some invariants about the shape of the data structures used and the consistency of the data stored in them. The invariants are given as informal comments in English. We encoded them as relational invariants in Alloy and checked whether or not our generated summaries are sufficient to show the preservation of these invariants. 6 invariants were checked in 15 procedures. Out of those 90 checks, 67 of them passed and the other 23 failed. The invariant with the highest failure rate is the one stating that the edge-set data structure contains all edges in the adja-

procedure	time (sec)	property
containsVertex(v)	0.1	no object fields are mutated
containsEdge(e)	0.1	no object fields are mutated
add(v)	0.1	v's final edge list is either empty or unchanged
		the edge lists of non-v vertices are unchanged
addEdge(e)	0.7	no new edges are added apart from e
		only adjacency lists of ends of e are changed
addEdge(v1, v2)	0.8	only adjacency lists of v1 and v2 are changed
		no edge is added except between v1 and v2
		final graph has an edge connecting v1 to v2
removeEdge(e)	0.5	only adjacency lists of ends of e are changed
		all edges but e remain in the graph
remove(v)	0.9	the final graph does not contain v
minSpanningTree()	2.7	all edges of MST belong to the original graph
		nodes with no edges are not added to MST
		edges of the original graph are unchanged

Table 7.4: Partial specifications checked in graph procedures.

cency lists of the vertices. In most of the procedures, the adjacency lists and edge set are modified within branches. Since the effects of different branches are unioned to form the final summaries, the summaries for such procedures allow any combination of modifications, and thus they are too weak to show the edge-set consistency invariant.

- *Post conditions.* Lacking formal specifications of the procedures, we checked the generated summaries for the 15 procedures against some partial post conditions. Table 7.4 shows a subset of the properties that the summaries were sufficient to check. The *time* column gives the time to generate a summary in seconds, and the *property* column is an English description of the checked partial post-condition.

The most important cases of information loss are from the following two cases:

(1) A procedure which returns a boolean value, as happens in `containsEdge` and `containsVertex`. Our technique unions the possible return values and reports that the return value is either true or false. The summary still provides a frame condition indicating which variables and fields are unchanged by the procedure,

but the information about the return value is not useful. The two such cases in this study have no side effects, so the summaries are the full frame conditions.

(2) A procedure with a loop that mutates the same field it is traversing, as in `remove`. Our technique is able to determine whether or not the resulting elements are a subset of the prior elements, but it typically cannot tell which is removed or if the order remains the same.

7.4.2 Effectiveness

In order to evaluate the effectiveness of procedure summaries that we generate, we compared Karun – which performs specification refinement starting from the initial summaries produced by our abstract interpretation technique – to another implementation that performs the same analysis, but starts from *frame conditions.* Frame conditions specify what fields and variables are not mutated by a procedure, but they provide no information about how the mutated fields and variables are modified. Although such summaries are simple and very cheap to compute, they may cause the specification refinement technique to go through many refinements in order to check a property.

Frame conditions cannot be computed statically with complete accuracy of course. Therefore, we compute conservative conditions: no frame condition is generated for a variable or field that *may* be modified at a call site. In order to take care of possible aliasings, if a procedure updates a field f of an object of type T, the generated summary allows f to change in all objects of type T.

In order to achieve a fair comparison, whenever possible, we shared code between Karun and the implementation based on frame conditions. That is, both implementations use the same translation of code to Alloy and solve the generated formulas using Kodkod. They both use ZChaff in the backend and unsat core to refine specifications. The comparison, therefore, shows the effects of the initial abstraction on the analysis performance.

We analyzed some procedures of the OpenJGraph and Quartz APIs using both

procedure	property	objects/ unrolls	frame conditions		abstract interpretation		speed-up
			time (sec)	refs	time (sec)	refs	
scheduleJob	returns valid date	3/3	189	12	11	5	17.2
add	vertex added	5/5	52	15	1	0	52
addEdge	other edges are unchanged	4/4	157	61	46	0	3.4
addEdge	an edge added b/w given nodes	6/6	30	5	1	0	30
minSpanTree	subgraph	5/5	3272	64	36	0	90.9
getTrigNames	all names returned	5/2	8	3	7	1	1.4
storeCalendar	triggers updated	4/1	38	47	22	31	1.7
storeJob	job is stored	7/1	399	65	149	12	2.7

Table 7.5: Experimental results: comparing abstract interpretation to frame conditions.

implementations. The results are given in Table 7.5. The `time` columns give the total analysis time, including the time spent on initial abstraction and all refinements, in seconds. The `refs` columns show the number of refinements needed to check the property in each case. The last column summarizes the speedup gained as a ratio between total analysis times using frame conditions and abstract interpretation.

As shown in this table, in most cases, using the abstract interpretation technique improves the analysis time substantially. This is because, in general, the summaries generated in this way are much more effective in ruling out invalid program executions. Since the initial abstraction contains fewer invalid executions, in general, fewer refinements are needed to check a property, and thus the analysis time is improved.

In a few cases, however, the gained speedup is close to one. That is, there is no considerable difference between the performances of the two implementations. This is because either the called procedures are so complex that the abstract interpretation cannot infer much more than frame conditions about their behaviors (e.g. the `storeCalendar` case) or they are so simple that even starting from frame conditions, the analysis can infer their behaviors in a few refinements very easily (e.g. the `getTrigNames` case).

Although computing initial summaries using the abstract interpretation technique takes more time than computing frame conditions, the resulting summaries improve the performance so much that the overhead of computing them is negligible compared to the total analysis time. In other words, our current experiments suggest that they are much more cost-beneficial than frame conditions. More experiments have yet to be done to evaluate the technique on procedures with deeply nested calls.

Chapter 8

Conclusion

This chapter first summarizes the thesis by describing the main ideas briefly, then compares our analysis technique to other related work, and finally concludes the thesis by discussing different aspects of our technique and highlighting some opportunities for future research.

8.1 Summary

This thesis described a framework for statically checking a given procedure against a property. The framework introduces an analysis that does not depend on any user-provided annotations beyond the property to check. It performs a modular analysis in which specifications of called procedures are inferred automatically from the code. In order to provide scalability to large programs, the specifications are inferred on demand: initial specifications are only rough over-approximations of the code; they are refined further only if checking the property requires more precise specifications. The analysis terminates when either the property is validated or a valid counterexample is found.

The framework is parameterized over some basic operations: a translation function that converts the code to a logical formula, a specification extraction function that extracts the initial specifications of called procedures, a solving operation that solves the generated logical formula, and a proof generation mechanism that computes a proof of unsatisfiability whenever the solved formula is unsatisfiable. Any instantiation of the framework must provide these operations.

The thesis also described a lightweight, flow-sensitive, context-sensitive technique for generating symbolic summaries of object-oriented procedures. These summaries that are expressed in Alloy, a relational first order logic with transitive closure, can be thought of as detailed frame conditions; they describe which memory locations might be changed and in what ways. The technique is a simple application of the abstract interpretation framework that generates approximations of procedure specifications, to be later refined if needed. To that end, it is focused on generating summaries that are safe (they describe a superset of possible program behaviors), fast to generate (the time to extract a summary is linear in the size of the procedure and the number of fields), and small (we are often able to concisely summarize the effect of a loop by using transitive closure).

Furthermore, the thesis described our instantiation of the framework that is implemented in a prototype tool called Karun. This instantiation provides a fully automatic technique for checking Java programs against data structure properties of the heap expressed in Alloy. It translates the code along with call sites' specifications to a boolean formula that is handed to a SAT solver for solution. The specification refinement process is implemented using the unsat core facility provided by the SAT solver. Since Karun's translation of Java to boolean requires bounding the number of loop iterations and the size of the analyzed heap, it can only check properties with respect to bounded domains. Any counterexample returned by Karun is a real bug in the code. Lack of a counterexample, however, does not constitute proof of correctness; a bug may be found if the analysis is performed with respect to more loop iterations or a bigger heap.

We evaluated our analysis technique by using Karun to check some correctness properties of two open source Java APIs, namely OpenJGraph and Quartz. The OpenJGraph package successfully passed all the checks. Analyzing Quartz, however, resulted in finding two previously unknown bugs. We reported the bugs to the developers and they subsequently fixed them.

Furthermore, we evaluated the performance of our analysis technique by comparing Karun to a simple analysis that inlines all procedure calls. The results suggest that our technique performs substantially better when the analyzed property is a partial specification of the analyzed code. However, when the code is checked against its full specification, a simple inlining may perform better.

We also evaluated our abstract interpretation technique that generates the initial summaries of called procedures against two criteria: accuracy and effectiveness. The results were very encouraging: in many cases, our summaries were accurate enough to prove some major properties of the corresponding procedures. Only in a few cases, was there a significant loss of information. Furthermore, the summaries rule out many invalid executions of the abstract program and thus reduced the number of refinements needed to check a property. These experiments suggest that our abstract interpretation technique represents a useful balance between tractability and accuracy. More experiments have yet to be done to evaluate the summaries generated for larger programs.

8.2 Related Work

8.2.1 Underlying Ideas

Our analysis technique is based on three main concepts: (1) performing a fully automatic counterexample-guided refinement of procedure specifications, (2) translating code to boolean formulas and analyzing them using a SAT solver, and (3) automatically extracting procedure summaries that can be used to compute the initial abstraction of code. In this section, we describe the work related to each of these concepts.

Counterexample-guided Abstraction Refinement

Using counterexamples to iteratively refine abstract models of a system was first introduced by Kurshan [36], and then appeared in a number of analysis techniques (e.g. [3, 11, 41]) that focused on checking finite systems. Clarke, et. al. [11], for example, used this framework to analyze programs represented by *labeled Kripke structures* with respect to properties expressed in a fragment of *ACTL**. A Kripke structure is a finite state machine in which each state has at least one transition, and a labeling function maps each state to a set of atomic propositions that holds in that state. ACTL* is a form of the computational tree logic (CTL), a temporal logic in which time is modeled using a tree structure (each moment of time has several possible futures). CTL* is an extension of CTL that allows path and state operators to be freely mixed. The ACTL* logic is a subset of CTL* that only allows universal quantifiers over paths. Formal definitions of Kripke structures and different temporal logics can be found elsewhere [59].

Clarke's analysis uses the transition blocks corresponding to the variables of a given Kripke structure to partition those variables into some equivalence classes and compute the initial abstraction function. This function is used to construct the initial abstract Kripke structure. Standard model checking techniques are used to check the abstract structure with respect to the given ACTL* property. If a spurious counterexample is found, the abstraction function is refined by further partitioning one of the equivalence classes. This technique is implemented in NuSMV and was successfully used to check a Fujitsu IP core design. However, it can only handle systems that can be described as a finite state machine.

To our knowledge, the software model checker SLAM [4] is the first tool that has applied the counterexample-guided abstraction refinement framework to a program with an infinite number of states. SLAM checks a given C program with respect to a *temporal safety property* without requiring any user-provided intermediate annotations. A temporal safety property constrains that the code contains no bad executions, in the form of a finite state machine. An example is a property specifying that the code does not release a lock without acquiring it first. Since verifying such properties in an arbitrary piece of code is undecidable, SLAM is not guaranteed to terminate. However, in practice, it always terminates successfully, meaning that either the property has been verified, or a nonspurious counterexample has been found.

SLAM performs the analysis by iteratively refining a *predicate abstraction* [25] of the code to eliminate spurious counterexamples. Its analysis consists of three components: (1) C2BP, which abstracts a given C program as a boolean program (a C program in which all variables are of boolean type) using a given set of predicates, (2) BEBOP, which model checks boolean programs, and (3) NEWTON, which discovers additional predicates to refine a boolean program.

Predicate abstraction consists of abstracting concrete states of a program based on the values of a set of predicates. C2BP computes the initial abstract program based on the predicates extracted from the property. For each statement, it computes weakest preconditions of the predicates and strengthens them to a combination of the available predicates. This requires a potentially exponential number of calls to a theorem prover (in the number of predicates). BEBOP then checks if the error state is reachable in the resulting boolean program by applying a data-flow analysis that computes the set of reachable states for each statement. It handles loops by computing a fixpoint over the set of states associated with each statement. These sets are represented using binary decision diagrams (BDDs). NEWTON uses verification condition generation to check the validity of any found counterexample in the original C program and generates new predicates in response to spurious counterexamples. These predicates are used by C2BP to compute a new abstraction and the process starts over.

BLAST [26] is similar to SLAM. It uses predicate abstraction and constructs a reachability tree which represents a portion of the reachable, abstract state space of the program. If an error node is reachable in the generated tree, the corresponding error is checked for validity. In case of invalidity, a theorem prover suggests new abstraction predicates which are used to refine the program. BLAST differs from SLAM in that it uses lazy abstraction and local refinements to achieve better performance.

Although SLAM and BLAST have been successfully used to analyze substantial programs, they both aim at checking temporal safety properties of the code and cannot handle the rich data structure properties that we target. Their analyses terminate rapidly for properties about control state transitions of the code, but not necessarily for data structure properties. In fact, because of lack of transitive closure, even their specification languages have limited support for expressing rich data structure properties.

SAT-based Analysis

Our method is inspired by Jalloy [58], a tool for checking data structure properties of Java programs. Its analysis process is very similar to ours: it translates Java code to an Alloy formula by bounding the number of loop iterations and the heap size. It then checks the generated formula with respect to a specification, also expressed in Alloy, using a SAT solver. Jalloy is different from our tool in that it inlines all called procedures whose specifications are not provided by the user. It has been applied successfully to some small Java programs including an implementation of a red-black tree. Although Jalloy can check very rich properties of complex data structures, the experiments show that inlining limits its scalability to large programs. This motivated us to develop a technique that automatically infers procedure specifications as needed in order to provide better scalability.

Saturn [62] is another SAT-based tool that checks C programs with respect to temporal safety properties. Similar to our tool, Saturn's translation to a boolean formula requires finitizing loop executions. Therefore, it is aimed at finding bugs rather than proving correctness. However, unlike our tool which is guaranteed to return only valid counterexamples, Saturn may generate false positives. Therefore, its returned counterexamples must be manually inspected for validity.

Saturn handles procedure calls by computing procedure summaries similar to a type signature that can be efficiently used for interprocedural analysis. A procedure summary represents the relevant behavior of a procedure by a finite state machine. Given a set of states, input predicates, and output predicates, a procedure summary is computed as a set of tuples $(o, P_{in}, s, P_{out}, s')$ denoting that the procedure causes a state transition in the object o (which is accessible by both the caller and the callee) from the state s to the state s' if the input predicates of the set P_{in} hold before the procedure call. If this state transition takes place, the output predicates of the set

P_{out} will hold after the procedure call. These summaries are computed by making several calls to a SAT solver: all possible state transitions are enumerated under all combinations of predicates, and their feasibility is checked using a SAT solver. The sets of states and input and output predicates of each called procedure are specific to the property being checked and have to be provided separately. Inadequate choice of predicates can cause the analysis to produce false alarms.

The automatic procedure summarization technique implemented in Saturn allows it to scale to large programs. In fact, it has been successfully used to detect memory leaks and erroneous sequences of locks and unlocks in a Linux kernel. However, the summarization technique used in Saturn is highly tuned for checking finite state machine properties and cannot be applied to the kind of data structure properties that we target.

Magic [8] is also a modular software analysis tool that checks C programs against finite state machine specifications using a SAT solver. The specifications are expressed as *labeled transition systems* (LTS), finite state machines that differ from Kripke structures in that their transitions are labeled by actions rather than their states being labeled by propositions. Magic uses LTS not only as a specification language, but also as an abstract domain to represent the behavior of the procedure being analyzed. Since a procedure might behave differently under different settings of its input parameters, Magic abstracts a procedure as a list $< g_1, L_1 >, \ldots, < g_n, L_n >$ where each L_i is an LTS and each g_i is a guard formula over the input parameters of the procedure that encodes the condition under which L_i is executed.

Abstract labeled transition systems are computed based on predicate abstraction using a theorem prover. A SAT solver is used to check whether an abstract LTS conforms to the LTS that represents the given property. Any found counterexamples should be checked for validity. To our knowledge, Magic performs the abstraction phase automatically, but requires the user's guidance to check the validity of found counterexamples. If a counterexample is invalid, the abstraction is refined, again based on the user's inputs. Although Magic provides a modular analysis, it assumes that the user provides specifications of called procedures; any procedure with no specification is inlined. That is, unlike our technique, it provides no mechanism for summarizing procedures automatically. Furthermore, since it only accepts finite state machines for properties, it cannot handle the kind of data structure properties that we do.

Automatic Specification Extraction

The problem of summarizing the behavior of a procedure has been widely studied. In fact, almost all analysis techniques that statically check a program against a property compute an internal abstract model that summarizes the behavior of the code. Some such techniques have already been discussed. Other techniques, including shape analysis and theorem prover-based analyses, will be discussed later. In this section, however, we describe some of the techniques that compute procedure summaries not only to be used internally in a specific analysis, but also to be output to the user in order to be used in other settings. These techniques are compared with our abstract interpretation technique that extracts procedure summaries. Although we use those summaries to compute an initial abstraction, they are, in fact, stand-alone summaries that can be used in a variety of settings.

Heap manipulating procedures are summarized from different aspects: pointer analysis techniques (see e.g. [9,50,61]), for example, summarize the pointer information of a given procedure. They compute the set of memory locations to which a reference variable can point at each control point of the program. Side-effect analysis techniques (e.g. [10,27,43,49,52]), on the other hand, conservatively compute the set of heap locations that may be mutated by a procedure. They typically use a pointer analysis to approximate the objects to which the pointers can point. Pointer and side-effects analyses are particularly useful in optimizing, parallelizing and analyzing programs.

Our abstract interpretation technique is similar to those analyses in that it seeks the same end, but by different means: it uses relational expressions to approximate the set of objects pointed to by a variable, and then uses them to safely identify the set of locations that may be mutated by a procedure. Our technique differs from side-effect analyses in that it computes both *may* side-effects, i.e. the memory locations that can be mutated in some program execution, and *must* side-effects, i.e. the memory locations that are mutated in all program executions. Side-effect analyses typically compute only may side-effects. Furthermore, our technique computes not only *what* locations can be mutated by a procedure but also *how* those locations are mutated. Therefore, our analysis aims at providing more precise information about the code.

Tkachuk and Dwyer used side effect analysis to generate summaries [55]. To check a property of a program unit more efficiently, they approximate the behavior of the rest of the program with summaries. As in our approach, they generate both may and must side effects and specify the mutations that can be performed by a procedure. However, our analysis is potentially capable of generating more precise summaries (1) by recording the history of field updates by symbolic override expressions, rather than conservatively updating the field in all objects that may be aliased, and (2) by exploiting loop conditions in abstracting loop bodies.

There are also some *dynamic* techniques that generate procedure summaries automatically. Unlike static techniques that only use the text of a program, dynamic techniques execute the code to analyze its behavior. Daikon [18], for example, detects likely invariants about programs by executing a program, which is instrumented to write out certain variable values at each program point, over a test suite. An offline analysis then processes these values for an extensive set of invariants at each program point and outputs the invariants that are true in all executions. The most substantial difference between Daikon and our technique is that Daikon's results are not sound. That is, although the invariants returned by Daikon are guaranteed to hold in all test cases of the test suite used, there is no guarantee beyond that; they do not necessarily hold in general.

8.2.2 Overall Goal

In this section, we describe the static analysis tools that focus on checking data structure properties of code. Many of them, however, can only handle a subset of the data structure properties that we target.

Theorem Proving

The extended static checker for Java (ESC/Java) [21] checks a Java program with respect to a user-provided property. Similar to our technique, it performs the analysis on a finitized code (in which loops are unrolled a finite number of times), but expects users to provide specifications for all reachable procedure calls. Users express both the property and the intermediate annotations in the ESC/Java annotation language which is very similar to the Java modeling language (JML) [38]. However, since JML is designed for full program specification and the ESC/Java annotation language is intended for only lightweight specifications (for example, null dereferences, array bounds errors, and deadlocks), the two languages have some differences.

ESC/Java analyzes the code for counterexamples using the Simplify theorem prover [15]. It translates the given program to Dijkstra's guarded commands [16], encoding the property as an `assert` command. It then computes weakest preconditions to generate verification conditions as predicates in a first order logic. The Simplify theorem prover is used to prove the result. Failed proofs are turned into error messages and returned to the user. Since Simplify may loop forever on some invalid formulas, its analysis time is bounded in advance. Therefore, it sometimes reports a counterexample that might, with more effort, have been shown to be invalid. Consequently, unlike our technique, ESC/Java can produce false counterexamples. JML provides a facility, namely *model fields* and *model methods*, for defining abstraction functions. Such abstractions can be defined in Alloy as well. Our analysis, however, does not currently handle properties in terms of abstraction functions. This capability is included in Forge [14] and will be incorporated into Karun in future.

The developers of ESC/Java report that although the tool has been successfully used to check a number of programs, the overhead of annotating all procedure calls is the main obstacle to using it for larger systems [21]. This inspired Houdini [22], the annotation assistant for ESC/Java, which extracts procedure annotations automatically from the code. Houdini works as follows. For each called procedure, it generates a set of candidate annotations based on some heuristics. For example, for each variable v of integer type, it generates a number of annotations of the form v *cmp expr* where *cmp* is an integer comparison operator and *expr* is either another integer variable/field or a constant. For a reference variable v, it generates an annotation of the form $v \; ! = null$, constraining v to be non-null.

Once a candidate set of annotations is computed, Houdini uses ESC/Java to verify or refute each annotation. Incorrect annotations will be removed from the candidate set and the remaining annotations will be checked again. This process is repeated until a fixpoint is reached. Once a consistent set of annotations is computed, Houdini calls ESC/Java for the last time in order to check the annotated program against the given property, and presents the output to the user.

Houdini is similar to our tool in that it extracts procedure summaries automatically from the code. However, unlike our technique that does not assume any specific structure of the inferred specifications, Houdini relies on some initial heuristics that provide a template for the annotations that are possibly of interest. Furthermore, Houdini's specification inference process is highly tuned for inferring simple data structure properties (e.g. whether a pointer is null or not), and cannot handle the kind of rich data structure properties that we target.

Inspired by ESC, Flanagan introduced a method [19] to check properties of code by translating it to a constraint logic program (CLP) [31]. Each procedure is translated by two relations: one describing the program states that may cause an assertion to be violated, and the other describing a relation between pre- and post-states when the procedure executes normally. Since all procedures are translated, there is no need for user-provided procedure specifications.

Because the resulting formula can become very large, it is solved iteratively using the abstraction refinement framework: the constraint logic query is abstracted as a boolean formula using predicate abstraction. If the boolean formula has a solution, the solution is converted to a trace of the constraint logic query and a proof generating theorem prover, e.g. Verifun [20], is used to check whether the trace is satisfiable or not. In case of unsatisfiability, the theorem prover generates a proof by calling a decision procedure for the appropriate domain, e.g. linear arithmetic, or the theory of arrays. The proof is used to infer additional predicates in order to refine the current abstraction. The process then starts over at the abstraction phase.

Unlike our technique that aims at finding bugs, Flanagan's method proves that the given property holds in all program executions. That is, its output is either a 'yes' indicating that the property has been proved, or a 'no' which means a non-spurious counterexample has been found. However, because the problem of checking a program against a given property is undecidable, his method is not guaranteed to terminate. Flanagan's analysis is similar to ours in that they both use the proof of unsatisfiability generated by a decision procedure (a theorem prover in his case, and a SAT solver in our case) to refine the abstract model. However, unlike our technique that translates the code on demand, his technique first translates the whole code into CLP and then checks its satisfiability iteratively. To our knowledge, this technique only targets simple properties similar to those checked by ESC/Java, and does not handle the kind of data structure properties that we check.

Hob [37] and Jahob [35] use theorem provers to analyze a program against a given data structure property. Hob assumes a simplified object-oriented language which is structured by modules, and allows a subset of Java constructs including field dereferences and dynamic object allocations, but not inheritance or polymorphism. It handles data structure properties that can be expressed by the boolean algebra of sets. That is, the properties can involve equality, containment, and disjointness of sets of objects. For example, in a minesweeper game, the property can constrain the set of exposed cells to be disjoint from the mined cells.

Unlike our technique, Hob aims at full verification of programs: it terminates either when the property has been proved or when a counterexample has been found. In order to provide scalability to large programs, Hob analyzes the code in a modular way, checking one procedure at a time. It provides a suite of static analyses and decision procedures, each suitable for checking a particular set of data structure properties. Although Hob can handle properties about the contents of data structures, its specification language is not as expressive as ours. It cannot handle, for example, the properties involving relationships between the objects (e.g. if a key object is linked to a value in a map data structure), or the ordering of objects (e.g. in a list data structure). Furthermore, although in certain cases, Hob can automatically produce loop invariants, in general, it requires users to provide specifications for all called procedures and loops reachable from the analyzed procedure.

Jahob is a successor of Hob that handles a richer subset of Java as well as richer specifications. Its specification language is a subset of the language of the Isabelle interactive theorem prover [46] that allows verification of data structure implementations based on singly and doubly-linked lists, trees with parent pointers, priority queues, and hash tables. Although Jahob can verify important data structures, it still cannot handle arbitrary shapes of objects that our technique targets. Furthermore, it still requires user-provided annotations in order to analyze procedure calls.

Shape Analysis

Shape analysis techniques (see e.g. [24, 44, 47, 51]) are in general, similar to our approach in that they target data structure properties of programs. That is, they check properties that constrain the shape of the objects in the heap of a program. However, unlike our technique that only aims at finding bugs in a bounded domain, shape analysis techniques aim at proving the correctness of properties in general. Consequently, they can typically analyze a restricted class of data structures, and cannot handle arbitrary linked data structures as we do. These techniques are traditionally difficult to scale to large programs because of their complexity and limited support for modularity.

Parametric shape analysis (PSA) [51], for example, encodes the heap of a program as a graph in which the nodes represent memory locations and edges represent field relations. Since these concrete graphs can be arbitrarily large, PSA conservatively abstracts them by merging all nodes equivalent under a particular set of shape predicates into a single node. By construction, the resulting abstract shape graph is guaranteed to have a bounded size.

There are two categories of shape predicates: core predicates, and instrumentation predicates. Core predicates are necessary for the analysis. They are either pointer variables treated as unary predicates, or pointer fields treated as binary predicates. Instrumentation predicates, however, are a means to fine tune the analysis. They can be provided by the user in order to compute more accurate abstract graphs. An example is a predicate specifying that all elements of a data structure are reachable from a root variable.

PSA evaluates predicates using Kleene's 3-valued logic [34]. In this logic, the value of each formula can be *true*, *false*, or *unknown*. Therefore, the output of checking a data structure property can be *true*, meaning that the property is proved to be correct, *false*, meaning that the property is known to be incorrect, or *unknown*, meaning that nothing can be deduced.

Parametric shape analysis has been instantiated in TVLA (three-valued logic analyzer) [39]. It takes the initial shape graphs as input, and performs an abstract interpretation based on the semantics of each statement in the program. It requires the user to specify how each statement affects each predicate of interest. TVLA has been successfully used to verify insertion sort and bubble sort [40] as well as some operations of mutable lists and binary search trees [48].

PSA performs a one-pass analysis whose precision depends on the user-provided set of predicates. Our technique, however, does not require any user-provided annotations. It can automatically refine the abstraction until a conclusive result is obtained. However, unlike PSA, our method cannot verify the property in general; it only performs a bounded verification aimed at finding bugs.

The pointer assertion logic engine (PALE) [44] is another tool for verifying data structure invariants. It can analyze the properties of all data structures that can be described by *graph types*. Given an annotated program, PALE translates the code and the annotations to formulas in monadic second order logic. These formulas are solved by the MONA decision procedure [33]. When the analysis terminates, either the invariants have been proved or a counterexample has been generated. Although the MONA logic is decidable, it has an inherent non-elementary worst-case complexity[1].

[1] A problem has *non-elementary complexity* if there is no algorithm for it whose runtime is bounded by $exp_k(n)$ for any k. The expression exp is defined as $exp_0(n) = n$ and $exp_k(n) = 2^{exp_{k-1}(n)}$.

PALE, however, has been shown to perform efficiently in practice.

A graph type is a data structure defined by records that have two kinds of fields: data fields which define a backbone tree structure, and pointer fields which can point to any node of that tree. A pointer field is annotated with an expression that specifies its destination. Examples of graph type data structures include trees with parent pointers, linked lists, and red-black trees.

In order to use PALE, the user annotates the program by providing pre- and post-conditions of called procedures, loop invariants, and data abstraction invariants. The annotations are expressed in pointer assertion logic (PAL) and are treated as hints for the underlying decision procedure. PAL is a monadic second-order logic expressed over records, pointers, and booleans.

Although PALE can verify a large class of data structures, it checks them one at a time. Therefore, it is not suitable for checking representation invariants between different datatypes of a program. Furthermore, the success of its analysis depends on the user-provided annotations. Our technique, however, does not require any user-provided annotations beyond the property to check, and can handle any number of data structures with arbitrary shapes. But it does not provide a general proof, only verifying the property in a bounded domain.

In order to reduce the amount of user-provided annotations, some shape analysis techniques use predicate abstraction to infer loop invariants. Bohne [60], for example, takes a set of abstraction predicates and performs a symbolic shape analysis on boolean heap programs [47]. A boolean heap is an abstraction of the heap in which the abstract state is represented by a *set* of bitvectors (rather than a single bitvector) over the abstraction predicates. Bohne uses a number of decision procedures to infer loop invariants that may contain reachability predicates, universal quantifiers, and numerical expressions. User-provided annotations are needed to hint on the predicates that should be used in the abstraction of each code fragment. The system then verifies that the inferred invariants are sufficient for proving the post-condition by generating weakest pre-conditions. Bohne has been successfully used to verify some operations of linked lists, trees, two-level skip lists, arrays, and sorted lists.

Lazy shape analysis [7] also incorporates predicate abstraction into shape analysis. It is implemented by extending BLAST [26] with calls to TVLA. The analysis starts with using the predicate *true* for predicate abstraction, and a trivial shape graph that represents all heaps, for heap abstraction. Upon finding spurious error paths, it refines the abstraction by using Craig interpolation [42] to discover additional predicates automatically. To improve performance, lazy shape analysis constructs the initial abstraction on-the-fly and refines it only at the necessary program locations.

The predicates inferred during refinement include both the ones used in predicate abstraction, and those used in the abstraction of the shape graphs. Rich predicates (i.e. the instrumentation predicates of TVLA) are inferred using pre-defined shape-class generators (SCG). An SCG is a function that takes as input the pointers and fields that should be tracked, and returns a set of predicates. Since the abstraction and its subsequent refinements are local, each program location has its own SCG. During the course of the analysis, an SCG might be shown to be insufficient, and thus be replaced by a finer SCG. If a finer SCG cannot be found, the algorithm terminates without being able to check the property.

Lazy shape analysis is similar to our technique in that it verifies a data structure property by iteratively refining the granularity of an abstraction. However, unlike our technique, it is not fully automatic yet; its refinement phase depends on a pre-defined, carefully constructed set of shape class generators that is provided as an input to the algorithm. If the algorithm terminates, it outputs a sound 'yes' or 'no' answer. Our technique, on the other hand, always terminates, but only performs verification with respect to a finite domain.

8.3 Discussion

8.3.1 Merits

We have presented a static program analysis technique that targets data structure properties of heap-manipulating programs. Several analysis techniques were developed to check this class of properties. Most of them, however, aim at *proving* correctness, and thus do not scale to large programs without extensive user-provided annotations. Most program analysis techniques that scale to large programs, on the other hand, cannot check data structure properties; they only target control-intensive properties described by finite state machines.

Our analysis represents a useful balance between tractability and scalability. It provides a fully automatic technique that potentially scales to large programs, but does not require any user-provided guidance or annotations beyond the property being checked. The technique, however, cannot prove correctness of properties; it only checks them in a bounded domain, looking for counterexamples.

Our analysis technique is modular. It checks a procedure against a property using the specifications of its called procedures as surrogates for their code. Although it infers the specifications of called procedures automatically and does not rely on user-provided annotations, it can benefit from such annotations whenever available. That is, the user can improve the analysis time by providing the specifications of some procedures. Our experiments exploited this; in checking Java APIs, we manually provided the specifications of the calls to the library procedures rather than having the tool infer them from their code.

The analysis framework introduced in this thesis provides a unified approach in which a constraint solver is used for both checking the property and refining specifications. Although our instantiation of this framework uses a SAT solver as constraint solver, any constraint solver capable of generating proofs of unsatisfiability can be used as the backend engine.

We also presented a lightweight static technique to extract procedure specifications automatically from their code. The technique generates summaries that describe how a procedure mutates the state of a program, by specifying its both may and must side-effects in a relational logic. Although used as part of our program analysis technique, this summarization technique can be applied in a variety of settings. To our knowledge, the summaries that we generate are more precise than the ones generated by other sound techniques available.

8.3.2 Limitations

What we presented in this thesis has a number of limitations. Our prototype tool, Karun, handles only a basic subset of the Java language. It currently does not support arrays, exceptions, multithreading, and any numerical expressions other than integer additions and subtractions. One-dimensional arrays, exceptions, and full integer arithmetic have been recently added to the Forge language and can be easily incorporated into Karun. However, multi-dimensional arrays, multithreading constructs, and non-integer numerical expressions remain topics for future research.

Our analysis is not complete. It verifies properties only with respect to finite heaps and bounded loops. Therefore, although all found counterexamples are valid, lack of a counterexample does not constitute a proof of correctness. In fact, if the bounds are too small, no execution of the code will be possible, and thus any property will be vacuously true. Increasing the analysis bounds can result in higher confidence in the analysis results, though it often increases the analysis time exponentially. Therefore, users have to make a trade-off between the length of the analysis time and their level of confidence in the results.

We assume that the property of interest is expressed in terms of the data structures used in the analyzed procedure. That is, analyzed properties are code-level specifications, not system-level requirements. This limitation implies that the tool is best-suited to be used by programmers, or people with enough knowledge about the implementation details of the system. Any high-level requirement of the system should be converted to a specification in terms of the code's datatypes before being checked by our tool. A methodology to perform this conversion was proposed elsewhere [53].

Our technique for extracting procedure summaries also has a number of shortcomings. Currently, branch points always introduce imprecision into our summaries since we need to account for arbitrarily complex conditions. However, many conditionals have simple conditions and can be precisely summarized with a relational expression, and there is no fundamental reason why our technique should not do so. Such an extension might permit us to produce precise summaries for simple conditional procedures such as `isEmpty` methods.

We have found that matching a common loop pattern (traversal of simple linked data structures) generates considerably more precise summaries at a very low cost. We expect that several more common but simple patterns could be similarly beneficial. One such pattern one can consider is for remove operations; their summaries are currently not very accurate, but they often have short, precise, relational specifications.

8.3.3 Future Directions

Experimenting with our analysis technique raised some interesting questions that are topics for future research. Karun currently checks the validity of a counterexample by checking all of its called procedures in the order in which they are actually called. This phase might be optimized by first computing some priorities for procedure calls based on their relevance to the analyzed property, and then checking procedure calls in that priority order. We can further optimize the analysis by using SAT solutions in which some variables are marked as *don't care*, and thus guaranteed to be irrelevant, and multiple unsat cores that eliminate several inconsistent values of a call site post-state in one iteration of the technique. Investigating these ideas is left as future work.

Currently, a major obstacle that limits the scalability of our technique is handling call sites that require dynamic dispatch. Forge handles dynamic dispatch by expanding each call site to statically invoke all methods that may be called at that site. While this approach allows the analysis to consider all possible method calls, it does not scale to large Java applications with nested dynamically dispatched calls. Exploring other possible approaches to alleviate this problem remains a topic for future research.

Our technique aims at analyzing single-threaded programs. Several techniques are available that analyze the behavior of multi-threaded code. To our knowledge, however, they all aim at finding deadlocks or race conditions. Developing a technique for checking data structure properties of concurrent programs is an open problem.

Karun checks programs with respect to a finite domain: the size of the heap and the number of loop iterations are both bounded by user-provided values. If Karun does not find a counterexample, it only means that the property has been verified with respect to the analyzed domain; nothing beyond that is guaranteed. An interesting question is to evaluate the level of confidence that this kind of analysis achieves. This might be done by calculating the number of objects of each type that are needed to exhaust all possible executions of a specific program. If such bounds were available, checking a property of the program with respect to those bounds would result in a complete analysis; equivalent to proving correctness. A method to compute such bounds for state-based Alloy models has been proposed before [54]. Exploring similar approaches to compute sufficient bounds for program-derived Alloy models remains a topic for future research.

Bibliography

[1] The OpenJGraph API. http://openjgraph.sourceforge.net/.

[2] The Quartz API. http://www.opensymphony.com/quartz/.

[3] F. Balarin and A. L. Sangiovanni-Vincentelli. An iterative approach to language containment. In *International conference on computer-aided verification (CAV)*, pages 29–40, 1993.

[4] T. Ball and S. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN Workshop on Model Checking of Software*, pages 103–122, 2001.

[5] M. Barnett, B. E. Chang, R. DeLine, B. Jacobs, and R. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *International symposium on formal methods for components and objects (FMCO)*, pages 364–387, 2005.

[6] M. Barnett, K. R. M. Leino, and W. Schulte. The spec# programming system: An overview. In *Construction and analysis of safe, secure, and interoperable smart devices*, pages 49–69, 2004.

[7] D. Beyer, T. A. Henzinger, and G. Theoduloz. Lazy shape analysis. In *International conference on computer-aided verification (CAV)*, pages 532–546, 2006.

[8] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *International conference on software engineering (ICSE)*, pages 385–395, 2003.

[9] B. Cheng and W. Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *ACM SIGPLAN conference on programming language design and implementation (PLDI)*, pages 57–69, 2000.

[10] J. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL)*, pages 232–245, 1993.

[11] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *International conference on computer-aided verification (CAV)*, pages 154–169, 2000.

[12] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[13] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL)*, pages 238–252, 1977.

[14] G. Dennis, F. S. Chang, and D. Jackson. Modular verification of code with SAT. In *ACM SIGSOFT international symposium on software testing and analysis (ISSTA)*, pages 109–120, 2006.

[15] D. L. Detlefs, G. Nelson, and J. B. Saxe. A theorem prover for program checking. Technical Report 178, Compaq SRC, 2002.

[16] E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall, Englewood Cliffs, NJ, 1976.

[17] N. Een and N. Sorensson. Translating pseudo-boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:1–26, 2006.

[18] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.

[19] C. Flanagan. Software model checking via iterative abstraction refinement of constraint logic queries. In *Workshop on constraint programming and constraints for verification*, 2004.

[20] C. Flanagan, R. Joshi, X. Ou, and J. B. Saxe. Theorem proving using lazy proff explication. In *International conference on computer-aided verification (CAV)*, pages 355–367, 2003.

[21] C. Flanagan, K. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. In *ACM SIGPLAN conference on programming language design and implementation (PLDI)*, pages 234–245, 2002.

[22] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *International symposium of formal methods Europe on formal methods for increasing software productivity*, pages 500–517, 2001.

[23] The JML Specifications for the Java Linked List Data Structure. http://www.eecs.ucf.edu/ leavens/JML-release/javadocs/java/util/LinkedList.

[24] R. Ghiya and L. Hendren. Is it a tree, a DAG, or a cyclic graph? In *ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL)*, pages 1–15, 1996.

[25] S. Graf and H. Saidi. Construction of abstract state graphs via PVS. In *International conference on computer-aided verification (CAV)*, pages 72–83, 1997.

[26] T. A. Henzinger, R. Jhala, R. Majumdar, G. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *International conference on computer-aided verification (CAV)*, pages 526–538, 2002.

[27] M. Hind and A. Pioli. Which pointer analysis should I use? In *ACM SIGSOFT international symposium on software testing and analysis (ISSTA)*, pages 113–123, 2000.

[28] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[29] D. Jackson. Automating first-order relational logic. In *ACM SIGSOFT conference on foundations of software engineering (FSE)*, pages 130–139, 2000.

[30] D. Jackson, I. Shlyakhter, and M. Sridharan. A micromodularity mechanism. In *ACM SIGSOFT conference on foundations of software engineering (FSE)*, pages 62–73, 2001.

[31] J. Jaffar and M. J. Maher. Constraint logic programing: A survey. *Journal of Logic Programming*, 19(20):503–581, 1994.

[32] N. D. Jones and F. Nielson. Abstract interpretation: a semantics-based tool for program analysis. *Handbook of logic in computer science (vol. 4): semantic modelling*, pages 527–636, 1995.

[33] N. Klarlund, A. Moller, and M. I. Schwartzbach. Mona implementation secrets. In *International conference on implementation and application of automata (CIAA)*, pages 182–194, 2000.

[34] S. Kleene. *Introduction to Metamathematics.* North-Holland Publishing Co., Amsterdam, 2nd edition, 1987.

[35] V. Kuncak. *Modular Data Structure Verification.* PhD thesis, EECS Department, Massachusetts Institute of Technology, 2007.

[36] R. P. Kurshan. *Computer-aided verification of coordinating processes.* Princeton University Press, 1994.

[37] P. Lam. *The Hob System for Verifying Software Design Properties.* PhD thesis, Massachusetts Institute of Technology, 2007.

[38] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06f, Department of Computer Science, Iowa State University, 1999.

[39] T. Lev-Ami. TVLA: A framework for kleene based logic static analyses. Master's thesis, Tel-Aviv University, 2000.

[40] T. Lev-Ami, T. Reps, M. Sagiv, and R. Wilhelm. Putting static analysis to work for verification: A case study. In *ACM SIGSOFT international symposium on software testing and analysis (ISSTA)*, pages 26–38, 2000.

[41] J. Lind-Nielsen and H. R. Andersen. Stepwise CTL model checking of state/event systems. In *International conference on computer-aided verification (CAV)*, pages 316–327, 1999.

[42] K.L. McMillan. Interpolation and sat-based model checking. In *International conference on computer-aided verification (CAV)*, page 113, 2003.

[43] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *ACM SIGSOFT international symposium on software testing and analysis (ISSTA)*, pages 1–11, 2002.

[44] A. Moller and M. I. Schwartzbach. The pointer assertion logic engine. In *ACM SIGPLAN conference on programming language design and implementation (PLDI)*, page 221231, 2001.

[45] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Design automation conference (DAC)*, pages 530–535, 2001.

[46] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[47] A. Podelski and T. Wies. Boolean heaps. In *Static analysis symposium (SAS)*, pages 268–283, 2005.

[48] J. Reineke. Shape analysis of sets. Master's thesis, Universitat des Saarlandes, 2005.

[49] A. Rountev. Precise identification of side-effect-free methods in Java. In *IEEE international conference on software maintenance (ICSM)*, pages 82–91, 2004.

[50] R. Rugina and M. Rinard. Pointer analysis for multi-threaded programs. In *ACM SIGPLAN conference on programming language design and implementation (PLDI)*, pages 77–90, 1999.

[51] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002.

[52] A. D. Salcianu and M. Rinard. Purity and side effect analysis for Java programs. In *International conference on verification, model checking and abstract interpretation (VMCAI)*, pages 199–215, 2005.

[53] R. Seater, D. Jackson, and R. Gheyi. Requirement progression in problem frames: Deriving specifications from requirements. *Requirements Engineering Journal*, 12(2):77–102, 2007.

[54] I. Shlyakhter, M. Sridharan, and D. Jackson. Analyzing distributed algorithms with first order logic. Unpublished manuscript, 2002.

[55] O. Tkachuk and M. Dwyer. Adapting side effects analysis for modular program model checking. In *ACM SIGSOFT conference on foundations of software engineering (FSE)*, pages 188–197, 2003.

[56] E. Torlak and D. Jackson. Kodkod: A relational model finder. In *International conference on tools and algorithms for construction and analysis of systems (TACAS)*, pages 632–647, 2007.

[57] R. Valle-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *Conference of the centre for advanced studies on collaborative research(CASCON)*, pages 125–135, 1999.

[58] M. Vaziri. *Finding Bugs in Software with a Constraint Solver.* PhD thesis, EECS Department, Massachusetts Institute of Technology, 2004.

[59] W. Visser and H. Barringer. CTL* model checking for SPIN. In *International conference on computer-aided verification (CAV)*, pages 316–327, 1999.

[60] T. Wies, V. Kuncak, K. Zee, A. Podelski, and M. Rinard. Verifying complex properties using symbolic shape analysis. In *Workshop on heap abstraction and verification*, 2007.

[61] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *ACM SIGPLAN conference on programming language design and implementation (PLDI)*, pages 1–12, 1995.

[62] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL)*, pages 351–363, 2005.

[63] L. Zhang and S. Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Conference on design, automation and test in Europe (DATE)*, pages 10880–10886, 2003.