Finding Bugs in Software with a Constraint Solver

by

Mandana Vaziri-Farahani

S.M. Computer Science
Massachusetts Institute of Technology, 1996

B.S. Electrical and Computer Engineering
Carnegie Mellon University, 1995

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 2004

Author .....
Department of Electrical Engineering and Computer Science
December 17, 2003

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

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

Finding Bugs in Software with a Constraint Solver

by

Mandana Vaziri-Farahani

S.M. Computer Science
Massachusetts Institute of Technology, 1996

B.S. Electrical and Computer Engineering
Carnegie Mellon University, 1995

Submitted to the Department of Electrical Engineering and Computer Science
on December 17, 2003, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

We present a static technique for finding bugs in object-oriented procedures. It is capable of checking complex user-defined structural properties – that is, of the configuration of objects on the heap – and generates counterexample traces with no false alarms. It is modular, requires no user-provided abstractions, and is fully automatic.

It is based on the Alloy modelling language and analyzer. The method relies on a three-step translation: from code to a formula in Alloy, which is a first-order relational logic, then to a propositional formula, and finally to conjunctive normal form. An off-the-shelf SAT solver is then used to find a solution that constitutes a counterexample.

Modularity comes at the price of intermediate specifications. To minimize such annotations, the analysis contains a suite of optimizations that allow checking larger procedures with fewer annotations. The optimizations are based on a special treatment of relations that are known to be functional, and target all steps of the translation to CNF. Their effect is demonstrated with a prototype tool that can handle a subset of Java, by analyzing real code.

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

Acknowledgments

Graduate school has been a wonderful, but challenging journey for me. As it comes to an end, I would like to thank the people who made that possible.

Words cannot express my gratitude to my advisor, Daniel Jackson. I thank Daniel for giving me the opportunity to be a member of his research group, and to pursue this work. I am very grateful for his unrelenting and unconditional support throughout my graduate studies, and for all his help, especially during more difficult times.

I thank Joan Wheelis wholeheartedly for her unbounded support, and for helping me to find my way in graduate school. I am extremely grateful to Christopher Crick for all his encouragement during the writing of this thesis. It would not have been possible without their help.

I have been privileged to be a member of the Software Design Group. I thank my fellow SDGers, Mana Taghdiri, Ilya Shlyakhter, Manu Sridharan, Ian Schechter, Sarfraz Khurshid, Greg Dennis, Tina Nolte, Emina Torlak, and Jonathan Edwards for many useful discussions. I thank my readers, John Guttag, Gerard Holzmann, and Martin Rinard, for many useful comments.

Finally, I thank my parents, Chahnaz and Faramarz Vaziri, for sharing their passion for learning with me, at an early age. I thank my father for teaching me to think independently, and my mother the value of perseverance. I dedicate this thesis to them.

This research was supported by grant 0086154 from the ITR program of the National Science Foundation.

"As it grew lighter, Elephant Island loomed up through the mist on our port hand and for various reasons, thenceforth became our goal."
Ernest Shackleton, 1916

Contents

List of Figures

Chapter 1

Introduction

Software systems are notoriously prone to bugs – deviations from their expected behavior. Bugs arise from programmer mistakes or omissions, and their effect can be benign or catastrophic depending on the domain of application.

The most common way to find bugs in practice is testing. It consists of executing the code for different inputs and observing the output against what is expected. Testing has several drawbacks. First, whole classes of cases may be missed, and bugs may easily go uncovered. Second, the number of test cases required may be prohibitively large. Consider for example, a telephony system that handles 20 features. The different combinations of all these features must be tested separately to ensure that their interaction is handled correctly. But this results in 2^{20} test cases!

A different approach to finding bugs is static analysis, which consists of analyzing the text of a program, without actually executing it. The most rudimentary static analyzers are type checkers, supported by common programming languages. They are extremely helpful at uncovering common programming mistakes, such as calling a procedure with the wrong number of arguments, but they can also sometimes help to catch algorithmic bugs. Type annotations are a form of specification, but they are limited in the kinds of properties they express.

Ideally, one would want to check arbitrary specifications statically. This is of course undecidable, so existing analyses make a tradeoff between level of automation and expressivity of the properties checked. At one end of the spectrum, program analysis techniques are geared at particular properties, such as for example detecting opportunities for program optimizations, but are fully automatic. At the other end, theorem proving methods can handle very rich specification languages, but are poorly automated.

By either giving up termination, or completeness, some methods allow richer properties to be checked automatically. In recent years, finite state verification techniques have proven to be useful for finding bugs in code. These methods consist of exploring all the possible executions of a fragment of code, and checking for behavioral properties. They typically output a counterexample when a property is found not to be satisfied. They often rely on an abstraction of the code, which renders it finite state and makes it manageable.

Finite state verification techniques have been shown to be very effective in checking

properties describing event sequences. Software systems differ from hardware in that they have a very rich structure, and these techniques are not aimed at handling structural properties readily. The specification languages are often not rich enough. Some techniques require the user to manually abstract aspects of the system. Others run into inefficiency or non-termination when dealing with data-intensive code.

This thesis presents a finite state verification technique for checking structural specifications of code. It is modular, fully automatic, and requires no user-provided abstractions. It can output counterexample traces when specifications are found not to be satisfied, and it has no spurious error reports.

1.1 A New Code Analysis

The analysis technique we present here is based on the Alloy [17] modelling language and its analyzer. It takes as input a Java procedure, a property to verify, expressed in Alloy, and upper bounds on the number of iterations per loop and on the number of heap cells per class. Its output is a counterexample trace, if the property is violated, and "inconclusive" otherwise. It offers the following benefits to its users:

- **Ease-of-use** It is a push-button tool, and does not require its users to know of its underlying set-theoretic notions. It does not assume a deep knowledge of Alloy. Its specification language is easy to use.
- **Modularity** It allows the user to replace method calls with specifications. This makes the analysis modular. Since in practice users do not like to write intermediate specifications, it offers a suite of optimizations that allows checking larger procedures with fewer annotations.
- **No User-Provided Abstraction** It does not require any abstractions from its user. Instead it considers bounded instances of programs, with a bounded heap and loops unwound.
- **Sound Counterexamples** When a property is found not to be satisfied, it outputs a counterexample, which is an execution trace of the code. There are no false alarms, preventing the user from having to manually prune scores of spurious error reports.

1.2 An Example

We illustrate our technique on an implementation of insertion in red-black trees.

1.2.1 Insertion in Red-Black Trees

Red-black trees are binary search trees whose nodes have an additional attribute, a color, which is either red or black. Each procedure manipulating red-black trees

must preserve two invariants regarding colors, that together maintain trees that are roughly balanced.

1. If a node is red, then both of its children are black.
2. All paths from the root to a node with at most one child have the same number of black nodes.

Figures 1-1 and 1-2 show an implementation of insertion in a red-black tree. This code is a close transcription of pseudocode presented in a popular algorithms textbook [6]. It contains two classes `RBNode` and `RBTree`, and procedure `RBInsert`, which performs insertion.

The procedure takes an integer `i`. It first makes a node out of the integer and inserts it in the tree by calling `TreeInsert`, a procedure for insertion in binary search trees. Once the new element has been inserted, the procedure must restore balance in the tree, which is done in the main while loop. The red-black tree is modified by calling `LeftRotate` and `RightRotate` repeatedly.

1.2.2 Specification

In addition to the fragment of code above, the user provides a specification file shown in Figure 1-3.

Specifications are written in a stylized version of Alloy. They consists of a series of function definitions, stating constraints on the states before and after the execution of the procedure to be checked (called `pre` and `post` states respectively). The function called `specification` defines the top-level constraint.

The variables available to the user for writing specifications are the parameters of the procedure to be checked and `This` if it is non-static, to indicate the receiver.

Most functions typically take a parameter of type `S` that is an enumerated type containing elements `pre` and `post`. These are used to indicate the state of a field. For example, `n.(left.pre)` denotes the object pointed to by the `left` field of `n` in the `pre` state.

The first property, `CorrectColors`, says that the children of a red node are black. It states that that all `RBNode`s `n` that are reachable from the `root` of `t` by following zero or more `left` or `right` fields in state `s`, have the property that if `n` is red, then both its children are black.

We represent booleans by sets, which when empty represent the value false, and true otherwise. The formula `some n.(isRed.s)` means `n.(isRed.s)` is a non-empty set, i.e. it is true. Similarly, the keyword `no` indicates the empty-set, or false.

The second property, `IsBalanced` says that all paths from the root to a node with at most one child have the same number of black nodes. It states that for all `RBNode`s `n1` and `n2` reachable from the root, if they both have at most one child[1] (indicated by

[1]In the original algorithm [6], trees have null leaves that are considered to be black. We do not have these; this is why we need the `HasAtMostOneChild` function.

```
class RBNode {
  boolean isRed; int key;
  RBNode right; RBNode left;
  RBNode parent;
  public RBNode(int i){
    isRed = false; key = i;
  }
}
class RBTree {
  RBNode root;
  void TreeInsert(RBNode z){
    RBNode k = null;
    RBNode x = this.root;
    while (x != null){
      k = x;
      if (z.key < x.key) x = x.left;
      else x = x.right;}
    z.parent = k;
    if (k == null) this.root = z;
    else if (z.key < k.key) k.left = z;
    else k.right = z;
 }
 void LeftRotate(RBNode z){
   RBNode y = z.right;
   z.right = y.left;
   if (y.left != null) y.left.parent = z;
   y.parent = z.parent;
   if (z.parent == null) this.root = y;
   else if (z == z.parent.left)
       z.parent.left = y;
   else z.parent.right = y;
   y.left = z;
   z.parent = y;
 }
```

Figure 1-1: Code for Insertion in Red-Black Trees - Part 1

```
void RBInsert(int i){
  RBNode h = new RBNode(i);
  this.TreeInsert(h);
  h.isRed = true;
  while (h != this.root &&
            h.parent.isRed == true){
    if (h.parent == h.parent.parent.left){
      RBNode y = h.parent.parent.right;
      if (y != null && y.isRed == true){
        h.parent.isRed = false;
        y.isRed = false;
        h.parent.parent.isRed = true;
        h = h.parent.parent;
      } else {
        if (h == h.parent.right) {
          h = h.parent;
          this.LeftRotate(h); }
        //h.parent.isRed = false; //bug seeded
        h.parent.parent.isRed = true;
        this.RightRotate(h.parent.parent);
      }
    } else { //same as above with
             // left and right inverted
    }
    this.root.isRed = false;
  }
}
```

Figure 1-2: Code for Insertion in Red-Black Trees - Part 2

```
fun CorrectColors(s: S, t: RBTree) {
  all n: t.(root.s).*(left.s + right.s) |
    some n.(isRed.s)  =>
      no n.(right.s).(isRed.s)
      && no n.(left.s).(isRed.s)
}
fun IsBalanced(s: S, t: RBTree){
  all n1, n2: t.(root.s).*(left.s + right.s) {
    HasAtMostOneChild(n1) && HasAtMostOneChild(n2) =>
      #{n:RBNode|n in n1.*(parent.s) && no n1.(isRed.s)}
    =
      #{r:RBNode|n in n2.*(parent.s) && no n2.(isRed.s)}
  }
}
fun HasAtMostOneChild(n: RBNode){
  no n.(left.pre) || no n.(right.pre)
}
fun specification() {
  Tree(pre, This) && CorrectColors(pre, This)
    && IsBalanced(pre, This) =>
      CorrectColors(post, This)
}
```

Figure 1-3: Specification for Insertion in Red-Black Trees

the calls to function `HasAtMostOneChild`), then the cardinality of the set consisting of all black nodes on the path from `n1` to the root is equal to the cardinality of the corresponding set for `n2`. The expression `#e` denotes the cardinality of `e`.

These properties are followed by the `specification` function, which expresses the top-level specification to be checked. It states that if `This` is a well-formed tree in the `pre` state (indicated by `Tree(pre, This)`, and properties `CorrectColors` and `IsBalanced` are satisfied in the `pre` state, then `CorrectColors` is also satisfied in the `post` state.

The definition of `Tree` – not shown in Figure 1-3 – expresses what it means for a set of `RBNode`s to form a valid tree. This kind of constraint is often needed when writing specifications for our analysis technique, because the tool does not restrict itself to trees or lists, but deals with general graph structures. If well-formedness constraints are omitted, it may thus find counterexamples with structures that do not satisfy basic properties expected by the user. Providing such constraints is not hard, because they often consist of generic properties.

1.2.3 Running the Analysis

If we seed a bug in the code as shown in Figure 1-2 (line commented `bug seeded`), and run our analysis technique with the given specification for 5 heap cells per class, and 5 iterations, we obtain a counterexample in about 20 seconds[2].

In Figure 1-4, the numbers on each node indicate the keys, and red nodes are shown in white. The box indicates a violation of `CorrectColors` in the `post` state. The counterexample goes through 5 iterations; first, the node with key `4` is inserted in the tree, then after a total of four left and right rotations, nodes `4` and `5` are consecutive and both red.

This example illustrates some of the features of our technique. The user did not have to write any intermediate annotations beyond the properties to verify. The counterexample is always an actual trace of the procedure: there are no spurious error reports.

1.3 How the Analysis Works

Our analysis technique is based on the Alloy Analyzer – a constraint solver. A formula encoding the data and control flow of a procedure is checked for consistency against a structural property. The encoding is done for a bounded instance of the code: loops are unwound up to a small number, and only a limited number of heap cells are considered. This is based on the observation that if a fragment of code does not satisfy some property, then a counterexample of small size can often be exhibited.

Figure 1-5 shows the architecture of the tool. It takes a Java procedure and translates it to Alloy, which is a first-order logic (FOL) with relations. The specification

[2]This experiment was done on a 1.1GHz PentiumIII with 640MB of memory, using BerkMin as the underlying SAT solver.

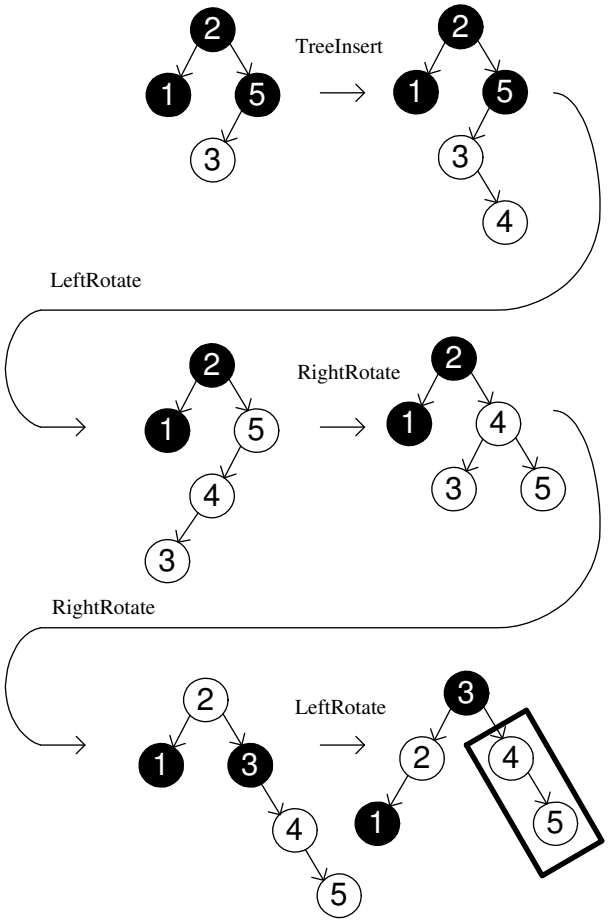

Figure 1-4: Counterexample for Insertion in Red-Black trees

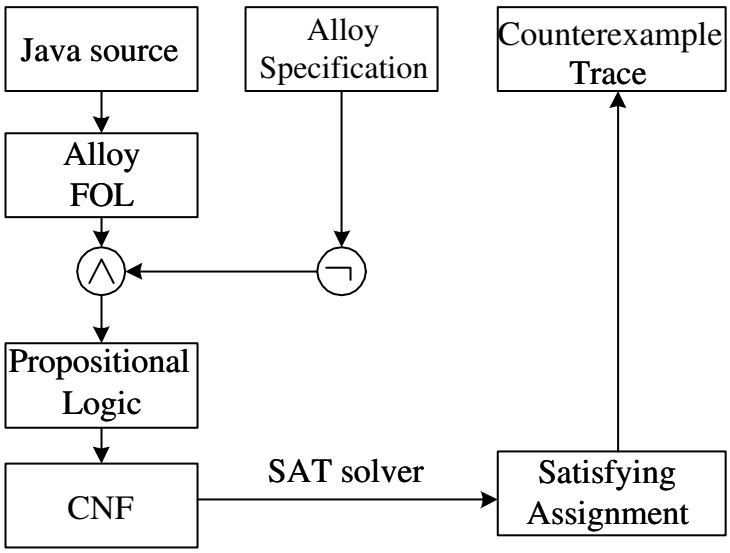

Figure 1-5: Architecture

is negated and conjoined with the formula corresponding to the code, yielding a formula for the problem. This has a satisfying instance if and only if there exists an execution trace of the code satisfying the negation of the specification, in which case a counterexample has been found.

In order to find such a satisfying assignment, the Alloy Analyzer first translates the problem formula to propositional logic, and then to conjunctive normal form (CNF). An off-the-shelf SAT solver is then used to find a satisfying assignment. If one is found then it is appropriately presented back to the user as a counterexample.

We can think of our analysis technique as a logical virtual machine: when it finds an instance, it is as if some execution trace of the code were followed. The advantage is that the trace is not actually executed sequentially, but obtained in a goal-oriented fashion to satisfy `pre` and `post` constraints. The analyzer can also come up with initial configurations that violate a specification.

1.4 Underlying Assumptions

Our analysis technique requires bounds from the user for the size of the heap as well as the number of iterations. An underlying assumption of our work is that checking exhaustively within small bounds can uncover most bugs. This is known as the *small scope hypothesis*. Consider, for example, checking that a `delete` procedure for linked lists is correct. It is probably sufficient to test it for all lists of size less than 3, beyond which the tests become redundant.

The small-scope hypothesis is only a hypothesis because of the undecidable nature of verification. We cannot therefore have an algorithm that would determine what bounds are sufficient for any given fragment of code. It is left to the user to select appropriate bounds for analysis.

There is more evidence that the small-scope hypothesis is suitable for designs than for code. A recent study [1] evaluates the small-scope hypothesis by measuring branch and statement coverage for a suite of procedures manipulating a variety of data structures. Coverage is a metric that is widely used to determine the efficacy of a test suite. The study shows that in most cases a bound of 7 heap cells provides complete coverage. As we shall see in a subsequent chapter, the experiments performed with our prototype tool scale to this bound. Checking exhaustively within small bounds is, however, better than a test suite with good code coverage, because it may uncover bugs due to missing code.

The small-scope hypothesis is suitable in software because of its inherent nature. In other engineering disciplines it can be acceptable to perform random testing – instead of testing all small cases, test randomly selected cases of arbitrary size. This can be useful because in other disciplines there is a "continuity" to the cases. For example, if we test that a bridge can support a certain weight, this implies that it can also support all smaller weights. In software, testing one case cannot be used to deduce anything about other cases: they have a "discrete" nature. Testing all small cases for structural properties has the effect of checking all possible "kinds of shapes" of the heap.

1.5 Contributions

The goal of this thesis is to promote the use of specifications for understanding programs and finding bugs, by providing an analysis technique that checks code against structural specifications, while minimizing user-intervention. More specifically, the objective is to check larger fragments of code with fewer annotations.

There are two avenues in which to approach the issue of checking larger procedures: top down and bottom up. In the top-down approach, we can view the constraint solver as a black box and explore abstractions relevant to the programming language at hand. In the bottom-up approach, we can look inside the constraint solver and optimize it for the purposes of verifying code. Ultimately, a combination of both avenues is ideal. This thesis however focuses on the bottom-up approach, by providing efficient encodings at all stages of translation from code to CNF. By reducing the size of the CNF produced, in terms of number of clauses and variables, the analysis technique runs faster and can handle larger fragments of code.

Our contributions are the following:

- An efficient encoding of Java in Alloy. Alloy provides several idioms for expressing object-oriented code. They are elegant and natural to use, but not efficient in practice: they run into the *scope-induced explosion* problem, where one or more types require a large bound, making the analysis intractable. We present an encoding that overcomes this issue.

- A compact encoding of fields in propositional logic. The constraint solver represents every entity as a relation, including Java fields. But these are mathematical functions and can be represented more compactly in propositional logic.

- Logical simplifications from propositional logic to CNF. When implemented alone, the compact field representation is not effective. A suite of synergistic optimizations are required to harvest its benefits. These include a series of logical simplifications that apply in the translation of propositional logic to CNF. They are inspired by the verification of code, but can be in fact used more generally in any context where they are applicable.

Together the optimizations reduce the size of the CNFs produced exponentially. Empirical results show improvement in the run-time of the analysis, as well as the ability of checking larger procedures. We present a series of case-studies to illustrate the ease of use of our technique, as well as the effectiveness of the optimizations.

1.6 Overview of the Thesis

Chapter 2 describes the Alloy modelling language, upon which this work is based. It provides the basis for understanding the subsequent encoding of Java in Alloy in Chapter 3, and illustrates some common idioms used to express object-oriented code. These idioms are elegant but are not efficient in practice. Chapter 3 presents an encoding that overcomes this issue. It further discusses how different aspects of the Java programming language can be handled by the analysis technique. Chapter 4 describes the translation steps from Alloy to CNF within the Alloy Analyzer, to prepare the reader for Chapter 5, which presents the optimizations. Chapter 6 presents a series of case studies to illustrate the effectiveness of the optimizations, as well as the ease of use of the technique. They include analyses performed with a prototype tool, of a procedure manipulating red-black trees, a garbage collection program, a fragment from a text processing system, and from the code of the prototype tool itself. We found real bugs in the last two case studies. Finally, Chapter 7 concludes and discusses related and future work.

Chapter 2

The Alloy Modelling Language

Alloy is a first-order relational logic for modelling software designs and their relevant properties and invariants. It has been used in a variety of case studies, such as the analysis of name servers [20], component frameworks [18], group key management schemes [39], and file synchronization [29], among others.

Alloy is a relational logic, in that expressions denote relations, and are composed together using relational operators. This facilitates describing structural properties. Indeed, the heap may be represented as a set of objects, and fields as relations among objects. A field in class C of type F can be modelled by a relation that maps C objects to F objects.

Alloy is a declarative language, meaning that behavior can be expressed by describing the relationship between the states before and after the execution of a procedure. Descriptions do not need to be operational, and this allows simpler, more concise specifications.

Alloy is amenable to automated analysis, and is equipped with a tool (the Alloy Analyzer) that generates instances and checks for consistency. Our analysis technique uses both the Alloy language and its analyzer. In a subsequent chapter, we will discuss optimizations for the analyzer that improve performance for the verification of Java code.

This chapter presents the Alloy language, and describes common idioms for modelling object-oriented code. These are elegant but inefficient. The next chapter presents an encoding of Java code that addresses this inefficiency.

2.1 Defining Types

We now present different aspects of the Alloy language by describing a model for a simple file system [35]. It consists of a set of objects that can be either files or directories. A filesystem has a root, a set of objects, and mappings for the contents of each directory as well as the parent directory for every object. We model and analyze a remove-directory operation.

```
module filesystem

-- File system objects
sig FSObject {}
part sig File, Dir extends FSObject {}

-- A File System
sig FileSystem {
  root: Dir,
  objects: set FSObject,
  contents: Dir ?-> FSObject,
  parent: FSObject ->? Dir
}
```

Figure 2-1: Signatures for a simple file system

2.1.1 Signatures and Atoms

Figure 2-1 shows some *signatures*, indicated with the keyword `sig`. A signature defines a type and a set of *atoms*. Atoms are similar to objects in object-oriented programs, in that they are the most basic fragment of the semantics of the language. But they differ from objects in that they are:

- *indivisible*: they cannot be decomposed further.
- *immutable*: they cannot be modified.
- *uninterpreted*: they do not have any underlying semantics.

In the example of Figure 2-1, we define the types `FSObject`, `File`, `Dir`, and `FileSystem`. `File` and `Dir` are subtypes of `FSObject`, as indicated by the keyword `extends`. This means that their atoms are contained in the set of atoms of their supertype. The keyword `part` means that `File` and `Dir` partition `FSObject`.

2.1.2 Fields

Once we have some types defined, we can introduce relations among them by defining *field*s. The type `FileSystem` has four fields: `root`, `objects`, `contents`, and `parent`. Field `Root` is the root directory; `objects` is a set of objects contained in the filesystem; `contents` is a mapping indicating the contents of each directory; and `parent` maps each object to its parent directory.

A field `F` in signature `S` denotes a relation `S -> F`, where `->` is the cartesian product of `S` and `F`, and consists of a set of tuples of atoms. Thus `root` and `contents` are relations of type `FileSystem -> Dir`, and `FileSystem -> Dir -> FSObject`, respectively.

The symbol `?` is a *multiplicity marking*, and imposes implicit cardinality constraints on a relation. The expression `Dir ?-> FSObject` states that for each atom of `FSObject` there is at most one `Dir`. We write `FSObject ->! Dir` to indicate that there is exactly one atom mapped to each `FSObject`.

```
module filesystem

-- File system objects
sig FSObject {}
part sig File, Dir extends FSObject {}

-- A File System
sig FileSystem {
  root: Dir,
  objects: set FSObject,
  contents: Dir ?-> FSObject,
  parent: FSObject ->? Dir
}

fact well-formedness {
  all fs: FileSystem {
    with fs {
      -- 1. root has no parent
      no (fs.root).(fs.parent)

      -- 2. objects are those reachable from the root
      fs.objects = (fs.root).*(fs.contents)

      -- 3. contents only defined on objects
      fs.contents in (fs.objects)->(fs.objects)

      -- 4. parent is the inverse of contents
      (fs.parent) = ~(fs.contents)
}
```

Figure 2-2: Well-formedness constraints for a simple file system

In the example, the type of `objects` is given by `set FSObject`, where `set` is another multiplicity marking, indicating that there are zero or more `FSObject`s for each `FileSystem`. To indicate that there is at most one, we use the keyword `option` instead. No marking in this case means exactly one. So there is exactly one root for each `FileSystem`.

2.2 Defining Facts

We continue the definition of our model of a simple file system by adding a set of constraints that describe some well-formedness conditions. These are introduced by the keyword `fact` in Figure 2-2. The lines of the `well_formedness` fact are implicitly conjoined together. In general, formulas enclosed by curly braces `{}` are implicitly conjoined. Comments begin with `--`.

Formula `1` says that the `root` of a filesystem has no parent, Formula `2` that the objects are those reachable from the root, Formula `3` that the `contents` of a filesystem are only defined on its `objects`, and Formula `4` that the `parent` relation is the inverse of `contents`.

2.2.1 Basic Expressions

Every expression in Alloy denotes a relation, i.e. a set of tuples of atoms. Sets are represented as degenerate unary relations, and scalars as singleton sets.
Basic expressions can be either:

- The name of a `sig`, denoting the set of atoms it represents,
- A field name,
- A quantified variable.

These basic expressions can be composed together with set operators, which apply to relations, since these are essentially sets of tuples of atoms:

- `+`: set union,
- `-`: set difference,
- `&`: set intersection.

2.2.2 Relational Join

The dot operator is a relational join and is defined as follows. Given two relations p and q of types $t_1 \to \cdots \to t_n$ and $t_n \to \cdots \to t_m$ respectively, $p.q$ is a relation of type $t_1 \to \cdots \to t_{n-1} \to t_{n+1} \to \cdots \to t_m$. A tuple $(p_1, \cdots, p_{n-1}, q_{n+1}, \cdots, q_m)$ is in $p.q$ if and only if there exists atoms p_n and q_n such that $(p_1, \cdots, p_n)$ is a tuple of p, $(q_n, \cdots, q_m)$ is a tuple of q, and $p_n = q_n$.

When p is a set, the join becomes relational image. In the example, `fs.root` is the relational image of `fs` under the `root` relation, i.e. the set of atoms that `fs` maps to under `root`.

The `with` construct is used to simplify Alloy formulas containing repeated applications of the dot operator to a relation, by factoring it out. For example,

```
with fs { no root.parent }
```

is syntactic sugar for

```
no (fs.root).(fs.parent).
```

A syntactic sugar, $p.q[s]$, denotes the expression $s.(p.q)$.

2.2.3 Relational Product

The arrow operator is relational product and is defined as follows. Given two relations p and q of types $t_1 \to \cdots \to t_n$ and $t_{n+1} \to \cdots \to t_m$ respectively, $p \to q$ is a relation of type $t_1 \to \cdots t_{n-1} \to t_n \to t_{n+1} \cdots \to t_m$. A tuple $(p_1, \cdots, p_n, q_{n+1}, \cdots, q_m)$ is in $p \to q$ if and only if $(p_1, \cdots, p_n)$ is a tuple of p, and $(q_{n+1}, \cdots, q_m)$ is a tuple of q.

The expression `(fs.objects)->(fs.objects)`, in Formula 3, denotes the relation that has a tuple `(o1,o2)` for every `o1` and `o2` in `fs.objects`.

2.2.4 Relational Inverse

The `~` operator is relational inverse. Given a relation p of type $t_1 \rightarrow \cdots \rightarrow t_n$, `~`$p$ is a relation of type $t_n \rightarrow \cdots \rightarrow t_1$. A tuple $(p_1, \cdots, p_n)$ is in `~`p if and only if $(p_n, \cdots, p_1)$ is a tuple of p.

In the example, `~(fs.contents)` in Formula 4 represents the inverse of `fs.contents`, mapping an object to the directory that contains it.

2.2.5 Transitive Closure

The symbol `^` denotes the transitive closure of a relation. Given a relation p of type $t \rightarrow t$, the transitive closure of p, consists of $p.p + p.p.p + \cdots$, where $+$ is set union. Another relational operator is the reflexive transitive closure of a relation, denoted by the symbol `*`. This is the same as the transitive closure, except that it also includes the relation itself.

In the example, `(fs.root).*(fs.contents)` denotes the set of all objects reachable from `fs.root` via the `fs.contents` relation.

2.2.6 Quantifiers and Logical Connectives

There are two ways of building basic formulas out of Alloy expressions. The first is set inclusion (`in`), and the second cardinality quantifiers:

- `no`: followed by an expression, means the expression represents the empty set,
- `some`: followed by an expression, means the expression has one or more elements.

Formula `1` says that the expression `(fs.root).(fs.parent)` is the empty set. Formula 3 states that `fs.contents` is contained in `(fs.objects)->(fs.objects)`, meaning that the `fs.contents` relation is only defined on `fs.objects`.

Alloy formulas are composed together using logical connectives:

- `!`, `not`: logical negation,
- `&&`, `and`: logical and,
- `||`, `or`: logical or,
- `=>`: logical implication,
- `<=>`: logical equivalence.

A special form of implication is `A => B, C`, and is equivalent to `A => B && !A => C`. Formulas also allow quantification:

- `all`: universal quantification,
- `some`: existential quantification.

In the example, variable `fs` is universally quantified.

2.3 Defining Functions and Assertions

It is sometimes useful to define parameterized formulas. These are introduced using the `fun` keyword, which stands for *function*. Figure 2-3 shows a function `rmdir`, which removes a directory from a filesystem.

Function `rmdir` takes three parameters, `fs` and `fs'` that represent a filesystem prior and after the operation respectively, and `d`, the directory to be removed. It first states some preconditions: `d` must be in the set of objects of `fs` other than its root, and `d` must not have any contents. The function defines contents of the new filesystem `fs'` to be the same as that of `fs` minus the mapping of `d`'s parent to `d`.

The definition of `rmdir` also states that the root of `fs'` is the same as that of `fs`. This is called a *frame condition*, which in general expresses what fields remain the same when some are modified. These conditions are necessary when a field is not constrained, in which case it could take on any value[1].

Following the function definitions in Figure 2-3 is an *assertion* introduced using the keyword `assert`. This is a consequence of the model that the user may wish to check using the analyzer. The assertion `rmdirRemovesOneDir` states that `rmdir` removes exactly the specified directory.

2.4 Running the Alloy Analyzer

In order to make Alloy amenable to automated analysis, the user must provide a bound on the number of atoms per type. This makes it possible to translate Alloy formulas to propositional logic, as we will see in Chapter 4. This bound is called *scope*.

In order to check assertion `rmdirRemovesOneDir`, we add the line:

```
check rmdirRemovesOneDir for 5
```

which directs the Alloy Analyzer to run with a scope of 5 atoms per type. For the assertion `rmdirRemovesOneDir` the Alloy Analyzer finds a counterexample: if `d` is the root of `fs`, then it is not removed and `fs` and `fs'` are identical.

2.5 Common Idioms for Object-Oriented Code

The next chapter describes how we encode Java procedures in Alloy. In this section, we prepare for that presentation by giving an overview of ways this may be done. Alloy is a flexible language and supports a variety of idioms for describing different kinds of systems. The concept of state, for instance, may be expressed in several ways, and this flexibility gives the user the freedom to express models in the most convenient way.

[1] In this example, the frame condition is not strictly required because the well-formedness constraints together imply that the root does not change.

```
module filesystem

-- File system objects
sig FSObject {}
part sig File, Dir extends FSObject {}

-- A File System
sig FileSystem {
  root: Dir,
  objects: set FSObject,
  contents: Dir ?-> FSObject,
  parent: FSObject ->? Dir
}

fact well-formedness {
  all fs: FileSystem {
    with fs {
      -- 1. root has no parent
      no (fs.root).(fs.parent)

      -- 2. objects are those reachable from the root
      fs.objects = (fs.root).*(fs.contents)

      -- 3. contents only defined on objects
      fs.contents in (fs.objects)->(fs.objects)

      -- 4. parent is the inverse of contents
      (fs.parent) = ~(fs.contents)
}

-- Delete the directory d
fun rmdir(fs, fs': FileSystem, d: Dir) {
  d in fs.(objects - root)
  no d.(fs.contents) -- d is empty
  fs'.contents = fs.contents - (d.(fs.parent))->d
  fs'.root = fs.root
}

-- rmdir removes exactly the specified directory
assert rmdirRemovesOneDir {
  all fs, fs': FileSystem, d: Dir |
    rmdir(fs, fs', d) => fs.objects - d = fs'.objects
}

check rmdirRemovesOneDir for 5
```

Figure 2-3: Functions and assertions for a simple file system

```
class ListElem {
  int val;
  ListElem next;
}
class List {
  ListElem first;
  static void swapTail(List l, List m){
    if (l.first != null && m.first != null) {
      ListElem temp = l.first.next;
      l.first.next = m.first.next;
      m.first.next = temp;
    }
  }
}
```

Figure 2-4: Code for the `swapTail` procedure

In this section, we present two common idioms for modelling objects on the heap. In the first [39], state is represented locally, and in the second [15] object references are modelled explicitly. These idioms result in elegant models, but their analysis is inefficient. Our encoding addresses this issue, at the cost of models that are harder to read. But since our analysis technique is automated, intermediate models need not be human-readable.

2.5.1 Local State Encoding

We illustrate these idioms by considering a procedure, `swapTail` (Figure 2-4), that takes two linked lists `l` and `m` and swaps their tails.

Since we are dealing with a logic, there is no built-in notion of state. Rather we need to encode it and the concept of location within the logic. One way of doing this is by having a special signature `State`. In the local-state idiom (Figure 2-5), fields are encoded as relations from `State` to a type. To express field dereferencing such as e.g. `l.first.next`, one writes

```
l.first[s].next[s]
```

where `l` is a `List`, `s` is a `State`, and the result of that expression is a `ListElem` object. Consider the assignment statement:

```
l.first.next = m.first.next
```

This gets encoded as:

```
l.first[s0].next[s1] = m.first[s0].next[s0]
```

In this formula, `s0` and `s1` denote the `pre` and `post` states of the assignment respectively. It says that in `s1`, the `next` field of the object denoted by `l.first[s0]` points to the object denoted by `m.first[s0].next[s0]`. This leaves the `next` fields of other objects, as well as the value of other fields, unspecified in `s1`. So we additionally need a series of frame conditions, marked `1`, `2`, and `3` in Figure 2-5. The first condition

```
sig State {}
sig ListElem {
  val: State -> int,
  next: State ->? ListElem
}
sig List {
  first: State ->? ListElem
}
fun swapTail(s0, s1, s2: State, l, m: List){
  some l.first[s0] && some m.first[s0] => { -- if
    some temp: ListElem {
      -- temp = l.first.next
      temp = l.first[s0].next[s0]
      l.first[s0].next[s1] = m.first[s0].next[s0]
      l.first[s0].val[s1] = l.first[s0].val[s0]     -- Frame Condition 1
      all o: ListElem - l.first[s0] |               -- Frame Condition 2
        o.next[s1] = o.next[s0] && o.val[s1] = o.val[s0]
      all o: List | l.first[s1] = l.first[s0]       -- Frame Condition 3

      -- l.first.next = m.first.next
      ...

      -- m.first.next = temp
      ...
    }
  }, { -- else
    s2 = s0
  }
}
```

Figure 2-5: Local Representation of State for the `SwapTail` procedure

states that the value of the `val` field does not change for `l.first[s]`, since we are only changing its `next` field. The second one says that for all `ListElem`s other than `l.first[s]`, their `next` and `val` fields remain the same. The last condition states that the `first` field remain the same for all `List`s.

For each statement, a new `State` atom is needed to hold the new state. The number of atoms required for `State` can therefore be very large, in the order of hundreds for a Java procedure, clearly beyond what the Alloy Analyzer can handle. This idiom therefore suffers from the scope-induced explosion problem, where one or more types require a large scope.

Moreover, frame conditions are not small: when updating one field, one must explicitly say that all other fields remain the same. The analysis would be more efficient if each update required fewer additional constraints.

2.5.2 Modelling Object References Explicitly

In the next idiom we discuss, object references are modelled explicitly (Figure 2-6). There are two main signatures `Object` and `Ref`. All classes are represented as subtypes of `Object`, and all class fields contain `Ref`s instead of other objects. For

example, `List` has a field `first` which is of type `ListElemRef` and not `ListElem` itself. The state of the heap is described by the signature `State`, which has two fields: a set of references, and a map from references to objects.

A special signature `NullRef` represents `null` in Java, and corresponds to the empty reference. To express field dereferencing such as e.g. `l.first.next`, one writes

```
l.(s.obj).first.(s.obj).next.(s.obj),
```

where `l` is a `ListRef`, `s` is a `State`, and the result of that expression is a `ListElem` object.

Having explicit references makes it easier to write frame conditions. These are now generic, and indicate what set of references map to the same objects in `obj`. Each assignment statement in `swapTail` is followed by a `modifies` constraint, which is a frame condition expressing what does not change as a result of the update. The body of `modifies` simply states that all the references not included in its `rs` parameter map to the same object in the post state. When updating a field, one no longer needs to write explicitly that all other fields remain the same.

On the other hand, this scheme also suffers from the scope-induced explosion problem. For each statement, a new `State` atom is needed, making its required scope very large. Another disadvantage is that the `obj` field in `State` is very large, since it maps all references, and it is expensive to have a universal quantifier over this relation every time there is an update.

2.5.3 A Look Ahead

The idioms described in the previous sections have their advantages. The explicit reference encoding makes frame conditions generic, simplifying modelling for the user. It also models `null` explicitly with a `nullRef` signature, and allows sets to contain `null` values. It is more expressive, in that sense, than the local-state encoding, or the one we present in the next chapter. It has been successfully used to model object interactions [15]. The local state encoding has been used for specifying a key management protocol [39], and is convenient when small traces (i.e. sequence of states) are sufficient. Both idioms are easy to read, but result in inefficient analysis when modelling large fragments of code, due to the scope-induced explosion problem.

In the next chapter, we will see an encoding that avoids the scope-induced explosion problem entirely, and requires small frame conditions. It is based on transforming the code to a form similar to Single Static Assignment (SSA) [4]. It results in Alloy models that are not very readable, but since our analysis technique is automated, it is not necessary for intermediate models to be human-readable.

```
sig Object {}
sig Ref {}
static disj sig NullRef extends Ref {}
sig NoRef extends Ref {}
fact { no NoRef }
sig State {
  refs: set Ref,
  obj: refs ->? Object
}
fact {
  all s: State, r: Ref | r = NullRef <=> no r.(s.obj)
}
disj sig ListElemRef extends Ref {}
disj sig ListElem extends Object {
  val: int,
  next: ListElemRef
}
disj sig ListRef extends Ref {}
disj sig List extends Object {
  first: ListElemRef
}
fun swapTail(s0, s1, s2: State, l, m: ListRef){
  l.(s0.obj).first != NullRef && m.(s0.obj).first != NullRef => { -- if
    some temp: ListElem {
      -- temp = l.first.next
      ...

      -- l.first.next = m.first.next
      l.(s0.obj).first.(s0.obj).next.(s1.obj) =
        m.(s0.obj).first.(s0.obj).next.(s0.obj)
      modifies(s0, s1, l.(s0.obj).first.(s0.obj).next) -- Frame Condition

      -- m.first.next = temp
      ...

    }
  }, { -- else
    modifies(s0, s2, noRef)
  }
}
fun modifies(pre, post: State, rs: set Ref){
  all r: pre.refs - rs | r.(pre.obj) = r.(post.obj)
}
```

Figure 2-6: Modelling Object References Explicitly for the `SwapTail` procedure

Chapter 3

From Java to Alloy

In the previous chapter, we saw that common Alloy idioms for expressing object-oriented code suffer from the scope-induced explosion problem, and large frame conditions. In this chapter, we present an encoding that overcomes these efficiency issues.

We first present the encoding for a small subset of Java, and then show how it may be extended to handle different features of the language: method calls, primitive types, arrays, subclasses, and exceptions. The chapter closes with a presentation of how we handle the Java API.

3.1 Basic Encoding

In this section we present the translation of a small subset of Java to Alloy. It includes: classes, objects, field dereferencing, assignments, if-then-else statements, return statements, and while loops.

3.1.1 Illustration

We illustrate our encoding of objects and the heap on a small example. Consider again the `swapTail` procedure reproduced in Figure 3-1, which takes two linked lists and swaps their tails. We use our analysis to check whether the `swapTail` procedure preserves the property that its inputs are acyclic. We write the Alloy specification shown in Figure 3-2. The function `specification` states that: if `l` and `m` are acyclic in the `pre` state, then `m` is also acyclic in the `post` state.

The auxiliary function `Acyclic` defines the constraint that a list is acyclic, by stating that for all `ListElem`s `e` reachable from `x.first` included, `e` is not reachable from itself.

When we run our prototype tool with 2 heap cells per type and 1 iteration, we obtain a counterexample (Figure 3-3). Black circles represent heap cells of type `List`, and white ones of type `ListElem`. These are labeled with atom names `LA0`, `LA1`, `EA0`, and `EA1`. Arrows represent fields. In the pre-state, list `m` is a list of one element, and `l` of two, and they share an element. In the post-state, `m` is a list with an element whose next field points to itself, and is therefore cyclic, violating the specification.

```
  class ListElem {
    int val;
    ListElem next;
  }
  class List {
    ListElem first;
    static void swapTail(List l, List m){
0    if (l.first != null
1              && m.first != null) {
       ListElem temp = l.first.next;
2      l.first.next = m.first.next;
3      m.first.next = temp;
4    }
  }}
```

Figure 3-1: Code for `swapTail` revisited

```
fun Acyclic(s: S, x: List) {
  all e: x.(first.s).*(next.s) | e !in e.^(next.s)
}
fun specification() {
  Acyclic(pre, l) && Acyclic(pre, m) => Acyclic(post, m)
}
```

Figure 3-2: Specification for `swapTail`

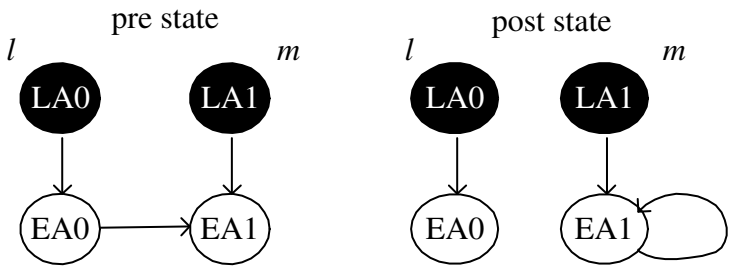

Figure 3-3: Counterexample for `swapTail`

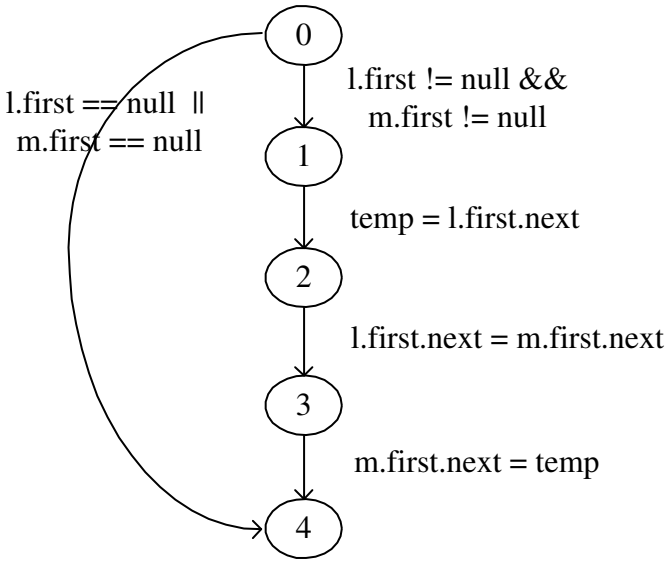

Figure 3-4: Computation Graph for `swapTail`

In the following sections, we will see how a procedure such as `swapTail` is encoded in Alloy. We translate a Java procedure into an Alloy model whose instances represent execution traces. We first construct a computation graph, then rename variables to encode state, and finally obtain formulas encoding the control and data flow.

3.1.2 Extracting a Computation Graph

To encode a Java procedure in Alloy we start by constructing a *computation graph*, which is a directed acyclic graph (DAG), whose nodes represent control points of the procedure, and whose edges are labeled with Java statements and conditions. The computation graph is essentially a standard control flow graph (CFG) where all loops have been unwound up to a parameter provided as input by the user. For example, one unrolling of:

```
a; while(p) s; b
```

gives the graph one would obtain as the standard CFG of:

```
a; if(p) s; assert !p; b.
```

The computation graph represents all the executions that terminate while going around the loops no more times than the parameter provided by the user for loop unwindings. Any other execution is not represented, and some errors may therefore be missed. But that is the tradeoff we make: our analysis technique cannot find all the bugs, but the ones it finds are not spurious.

Figure 3-4 shows the computation graph for the `swapTail` procedure. The edge between nodes (0) and (4) is traversed when the condition of the if statement is false.

```
if (n > 0)
   n  = n + 1
n = 2*n
```

Figure 3-5: Small fragment of code

```
if (n0 > 0)
   n1  = n0 + 1
n2 =  φ(n0, n1)
n3 = 2*n2
```

Figure 3-6: Small fragment of code in SSA form

3.1.3 Variable Renaming in the Computation Graph

To avoid scope-induced explosion and having a `State` signature as in the idioms of the previous chapter, our solution consists of renaming variables and fields when they are updated. For this reason, we call our encoding *name replication*. We use a technique similar to single-static assignment (SSA) [4] used in compiler optimizations. The next two sections explain both SSA, and our variant of it.

Single Static Assignment

SSA makes dataflow information explicit in a CFG by having multiple instances of the same variable. In every assignment to a variable `v`, `v` gets a fresh name, and every use of `v` is renamed to reflect the instance whose definition covers that control point. Thus a fragment of code is in SSA form if every use of a variable can be traced back to a single definition.

Consider the sample code shown in Figure 3-5. The first assignment to `n` gets a fresh name `n1`. Assume the instance of `n` prior to this fragment of code is `n0`. The use of variable `n` in the second assignment can be either renamed to `n0` or `n1`, and the question is how does that get resolved.

SSA solves this issue by introducing ϕ-functions as shown in Figure 3-6. These are inserted at join points as a way of unifying instance names. A ϕ-function for variable v is needed at a join point, if there are two or more incoming paths containing assignments to v. It has a parameter v_i for each of these paths, and its meaning is the ith parameter if the corresponding incoming path is executed. In the example, ϕ`(n0, n1)` is either `n0` or `n1` depending on which path is executed.

Name Replication

We rename variables as in the SSA form, but we also rename fields when they are updated. This will help curb the scope-induced explosion problem, as we shall see in Section 3.1.9. We need the renaming to be such that no path in the computation graph has two updates to the same variable or field instance. So parallel paths may share names, and we reuse them whenever possible.

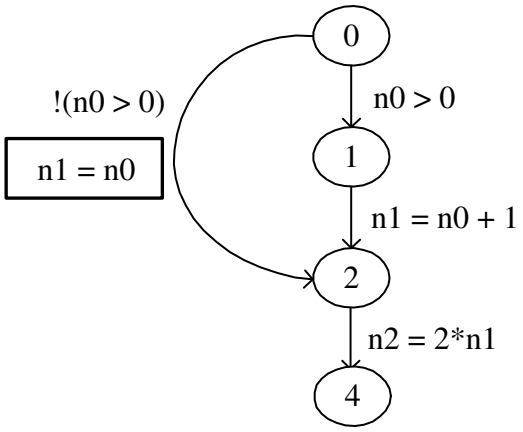

Figure 3-7: Renamed computation graph for small fragment of code

Renaming is done by providing an index for each variable at each node in the computation graph. In what follows we use v^i to denote the name of variable v at node i. Figure 3-7 shows the renamed computation graph for the small fragment of code of Figure 3-5. In this example, we have $\texttt{n}^1 = \texttt{n0}$ and $\texttt{n}^2 = \texttt{n1}$.

We do not use ϕ-functions, but instead attach frame conditions to edges in order to unify names at join points. If the name of a variable or field is the same on two joining branches, then the frame condition is not needed. In Figure 3-7, the box shows the additional frame condition: `n1` is set to be equal to `n0`, since `n` was not modified on that path.

In general, when an edge connects nodes i and j that assign a different index to a field or variable v, but v is not modified by the statement associated with the edge, we produce the frame condition:

$$v^j = v^i$$

Figure 3-8 shows the renamed computation graph corresponding to the `swapTail` procedure. The additional frame conditions are `temp1 = temp0` and `next2 = next0`, since `temp` and `next` are not updated on that branch.

3.1.4 Encoding the State

Given a renamed computation graph, we model the state in Alloy as follows. A field of type t appearing in class c is represented by a partial function from c to t for each program point. Local variables and formal parameters are modelled as optional singleton sets.

The `null` value in Java is represented as an empty set in the case of variables[1]. Moreover the field of an object is `null`, if its corresponding partial function does not

[1]This representation of null is convenient, but has the undesirable side-effect that Java collections may not include a null value.

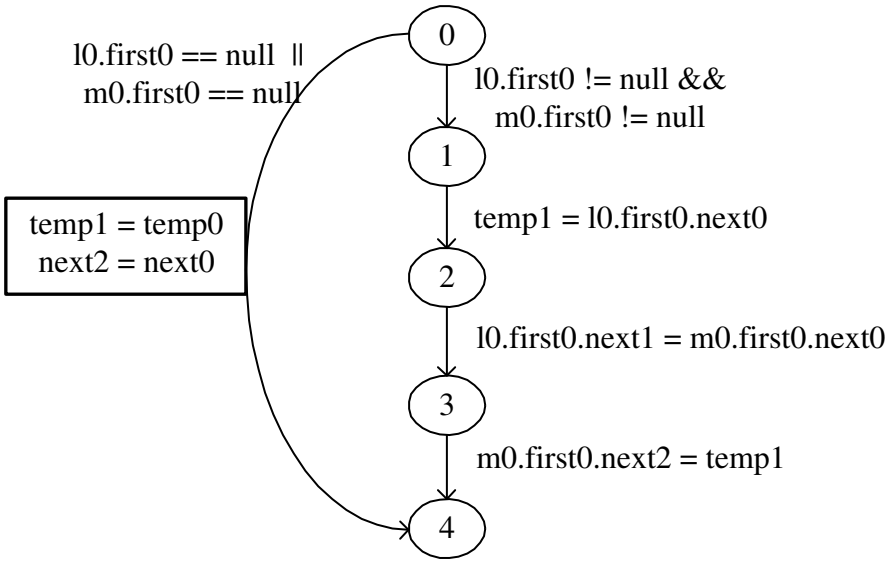

Figure 3-8: Renamed Computation Graph for `swapTail`

```
static sig state {
  first0             : List ->! ListElem
  next0, next1, next2: ListElem ->! ListElem
  val0               : ListElem ->! int
  l0, m0             : option List
  temp0, temp1       : option ListElem
  E_01, E_12, E_23, E_34, E_04: option Bit
}
```

Figure 3-9: Modelling state for the `swapTail` procedure

map that object.

The state encoding for `swapTail` is shown in Figure 3-9. It is encapsulated in a signature, `state`, which is static, meaning that it only has one atom.

Naming Edges

Additionally, the state includes a boolean variable for each edge, which is true if and only if that edge is traversed in an execution. This facilitates the encoding for the control and data flow, but it also serves as useful feedback to the user. When an instance is found, the edge variables that are true indicate what path was taken.

Booleans are represented by the type `Bit`, which has a single atom. A variable of this type may either contain that atom or not. If it does, it represents the boolean value `true`, and `false` otherwise.

Figure 3-9 shows the edge variables for `swapTail`. We write E_{ij} to denote the variable for the edge from node i to node j. For example, E_{23} represents variable `E_23` in this figure.

```
E_01 || E_04 &&
E_01 => E_12 &&
E_12 => E_23 &&
E_23 => E_34
```

Figure 3-10: Control flow encoding for the `swapTail` procedure

3.1.5 Encoding Control Flow

Now that we have an encoding for the state, we give Alloy formulas that represent the control and data flow of the procedure. The control flow is given by a formula that captures when an edge is traversed. For each node i, let $in(i)$ be the set of nodes having an outgoing edge to i, and $out(i)$ the set of nodes having an incoming edge from i. For each node i, we produce[2]:

$$\vee\{E_{ji} | j \in in(i)\} \Rightarrow \vee\{E_{ik} | k \in out(i)\}$$

These mean that if some node's incoming edge is traversed then some of its outgoing edges are also traversed. The formula encoding the control flow is the conjunction of these formulas.

For the first node in the computation graph ($i = 0$), where $in(i)$ is empty, we also conjoin the formula:

$$\vee\{E_{ik} | k \in out(i)\}$$

Infeasible paths are ruled out because some of the edges are labeled with control predicates, and these appear in the formula that encodes the data flow presented below. Note that if more than one outgoing edge is traversed, the constraint solver may generate an instance corresponding to more than one execution. But all these executions are feasible. Which one is presented to the user is a question of tool design. In the case of `swapTail`, the formula encoding control flow is shown in Figure 3-10.

3.1.6 Encoding Data Flow

We encode the data flow for each edge with a formula that indicates how variables are related before and after the execution of the statement corresponding to that edge. For each edge e from node i to j, we produce a formula:

$$E_{ij} \Rightarrow t$$

where t is the translation of the Java statement corresponding to e into Alloy. These mean that whenever e is traversed, the effect of the Java statement encoded by t is observed. The formula that encodes the data flow is then the conjunction of these formulas.

[2]The construct $\vee\{\cdots\}$ denotes the disjunction of a set of formulas.

$\mathcal{S}$: JavaStatement → Node → Node → AlloyFormula
$\mathcal{P}$: JavaPredicate → Node → Node → AlloyFormula
$\mathcal{E}$: JavaExpr → Node → AlloyExpr

1. $\mathcal{S}[v = \text{null}]\ i\ j \quad \equiv \quad \text{no } v^j$

2. $\mathcal{S}[v = e]\ i\ j \quad \equiv \quad v^j = \mathcal{E}[e]\ i$

3. $\mathcal{S}[e.f = \text{null}]\ i\ j \quad \equiv \quad \text{no } (\mathcal{E}[e]\ i).f^j$
   $\&\&\ (all\ o : O - (\mathcal{E}[e]\ i) \mid o.f^j = o.f^i)$

4. $\mathcal{S}[e.f = t]\ i\ j \quad \equiv \quad (\mathcal{E}[e]\ i).f^j = \mathcal{E}[t]\ i$
   $\&\&\ (all\ o : O - (\mathcal{E}[e]\ i) \mid o.f^j = o.f^i)$

5. $\mathcal{S}[\text{return } e]\ i\ j \quad \equiv \quad \text{result } = \mathcal{E}[e]\ i$

6. $\mathcal{P}[e_1 == \text{null}]\ i\ j \quad \equiv \quad \text{no } \mathcal{E}[e_1]\ i$

7. $\mathcal{P}[e_1 == e_2]\ i\ j \quad \equiv \quad \mathcal{E}[e_1]\ i = \mathcal{E}[e_2]\ i$

8. $\mathcal{P}[!p]\ i\ j \quad \equiv \quad !(\mathcal{P}[p]\ i\ j)$

9. $\mathcal{P}[p\ \&\&\ q]\ i\ j \quad \equiv \quad \mathcal{P}[p]\ i\ j\ \&\&\ \mathcal{P}[q]\ i\ j$

10. $\mathcal{E}[v]\ i \quad \equiv \quad v^i$

11. $\mathcal{E}[e.f]\ i \quad \equiv \quad (\mathcal{E}[e]\ i).f^i$

Figure 3-11: Translation rules for encoding the data flow

```
E_01 => some l0.first0 && some m0.first0
E_04 => no l0.first0 || no m0.first0
E_12 => temp1 = l0.first0.next0
E_23 => l0.first0.next1 = m0.first0.next0 &&
          all o: ListElem - l0.first0 | o.next1 = o.next0
E_34 => m0.first0.next2 = temp1 &&
          all o:ListElem-m0.first0 | o.next2 = o.next1
```

Figure 3-12: Data flow encoding for the `swapTail` procedure

The translation rules for Java statements are given in Figure 3-11. The translation function $\mathcal{S}$ is applied to Java statements, $\mathcal{P}$ to the predicates of branching statements, and $\mathcal{E}$ to expressions.

Recall that `null` is represented as the empty set. The first rule gives the translation for assigning `null` to a variable. The instance of that variable corresponding to the node after execution of the assignment (j) is equated to the empty set. Rule 2 is for the case when we are assigning an expression that is not the `null` literal.

Rule 3 and 4 give the translation for updating a field. Rule 4 equates the value of the expression t at node i with the value of the field f at node j of the object denoted by e at node i. It also adds a frame condition saying that the f field of no other object changes.

Rule 5 gives the translation of a `return` statement. A special Alloy variable `result` is used to hold the return value. The control flow resulting from a `return` is also handled in the computation graph. Rules 6 through 9 give the translation of Java predicates. The figure does not show logical Or, but its rule is similar to logical And. Finally rules 10 and 11 translate Java expressions: a variable becomes one appropriately renamed, and a field dereference becomes a relational image with a renamed field.

Figure 3-12 shows the data flow encoding for the `swapTail` procedure. The frame condition

```
all o:  ListElem - l0.first0 | o.next1 = o.next0
```

means that for all `ListElem`s other than `l0.first0`, the next relation remains the same.

3.1.7 Frame Conditions

Recall that renaming the computation graph also had the effect of adding frame conditions to some of the edges. When an edge from node i to j is also labeled with a frame condition of the form $v^j = v^i$ for some variable v, we produce the formula:

$$E_{ij} \Rightarrow v^j = v^i$$

In the case of `swapTail`, the additional frame condition is

```
E_04 => temp1 = temp0 && next2 = next0
```

```
fun Acyclic(s: S, x: List) {
  all e: x.(first.s).*(next.s) | e !in e.^(next.s)
}
fun specification() {
  Acyclic(pre, l) && Acyclic(pre, m) => Acyclic(post, m)
}
```

Figure 3-13: Specification for `swapTail` - revisited

In summary, there are two kinds of frame conditions in our encoding. The first is the one we just described. The second appears after every update of a field, and concerns only that field. Thus our frame conditions are small for updates, and possibly not at join points: they may involve many variables and fields. To alleviate this problem we do not generate frame conditions at joins for some variables whose values are not needed beyond that point in the computation graph. In practice, this optimization improves the efficiency of the analysis.

3.1.8 Putting It All Together

We now explain how a procedure's encoding is combined with the specification file. We reproduce the specification for `swapTail` in Figure 3-13.

For the convenience of the user, our technique defines `pre` and `post` fields for each class, to hide the indices in the computation graph. The glue between the specification file and the encoding for the procedure is shown in Figure 3-14. Signature `S` has two disjoint subsignatures `pre` and `post`, which are used to parameterize fields. The signature `List`, for example, has a field `first` of type `ListElem -> S`. So `l.(first.pre)` denotes accessing the `first` field of `List l` in the `pre` state.

The signature definitions are followed by a fact that makes the correspondence between these fields and the ones declared in the state. For example, it states that `next.post` is the same as `next2`. The `$` symbol is used to specify a field: `List$first` denotes the `first` field of `List`.

To allow the user to talk about the parameters of the procedure to be verified, we add the variables `l` and `m`, of type `List -> S`, to the `state` signature. We set these equal to `l0` and `m0` appropriately.

Figures 3-15 and 3-16 put together all the pieces of the encoding for `swapTail`, as well as its specification and the constraints needed to connect the two. The notation `with state` is an Alloy shorthand that helps to avoid preceding all fields with `state`. The Alloy function named `assertion` conjoins the encoding and the negation of the specification.

The last line in Figure 3-16 asks the Alloy Analyzer to produce an instance for `assertion` given a scope of 2 for all signatures. The result is the counterexample we saw before, reproduced in Figure 3-17.

```
sig S {}
disj static pre extends S {}
disj static post extends S {}

sig ListElem {
  val: int -> S,
  next: ListElem -> S
}

sig List {
  first: ListElem -> S
}

fact {
  List$first.pre = state$first0
  List$first.post = state$first0
  ListElem$next.pre = state$next0
  ListElem$next.post = state$next2
  ListElem$val.pre = state$val0
  ListElem$val.post = state$val0
  state$l = state$l0
  state$m = state$m0
}
```

Figure 3-14: Auto-generated Alloy signatures for connecting the specification to the procedure's encoding

3.1.9 Advantages over Common Idioms

No scope-induced explosion

The name replication encoding avoids types that require large scopes, by introducing more variables. There is no explicit representation of state objects. These are eliminated by introducing different instances of variables and fields.

To see how, consider a variable `v` of type `V`. In the local-state idiom, `v` is represented with a relation `v: State -> V`. To express the value of `v` in a state `s1`, one writes `v[s1]`. In the name replication encoding, `v` simply gets renamed to `v1`, and there is no longer a need to represent state `s1` explicitly.

Our encoding therefore does not run in the scope-induced explosion problem, which results from representing state objects explicitly. The tradeoff is that it requires more Alloy variables. This is not a problem, however, because its representation in propositional logic is more compact than the ones for the common idioms, as we shall see shortly.

Small frame conditions

Frame conditions are additional constraints that are added to a model. They are more expensive to analyze if they involve more formulas. This can be the case if there is a universal quantification over a type with a large scope. This is because the

```
sig Bit {}

sig S {}
disj static pre extends S {}
disj static post extends S {}

sig ListElem {
  val: int -> S,
  next: ListElem -> S
}

sig List {
  first: ListElem -> S
}

fact {
  List$first.pre = state$first0
  List$first.post = state$first0
  ListElem$next.pre = state$next0
  ListElem$next.post = state$next2
  ListElem$val.pre = state$val0
  ListElem$val.post = state$val0
  state$l.pre = state$l0
  state$l.post = state$l0
  state$m.pre = state$m0
  state$m.post = state$m0
}

static sig state {
  first0             : List ->! ListElem
  next0, next1, next2: ListElem ->! ListElem
  val0               : ListElem ->! int
  l0, m0             : option List
  l, m               : List -> S
  temp0, temp1       : option ListElem
  E_01, E_12, E_23, E_34, E_04: option Bit
}
```

Figure 3-15: Alloy model for the `swapTail` procedure - Part 1

```
fact encoding { with state {
  //Control Flow
  E_01 || E_04
  E_01 => E_12
  E_12 => E_23
  E_23 => E_34

  //Data Flow
  E_01 => some l0.first0 && some m0.first0
  E_04 => no l0.first0 || no m0.first0
  E_12 => temp1 = l0.first0.next0
  E_23 => l0.first0.next1 = m0.first0.next0 &&
            all o: ListElem - l0.first0 | o.next1 = o.next0
  E_34 => m0.first0.next2 = temp1 &&
            all o:ListElem-m0.first0 | o.next2 = o.next1

  //Additional Frame Condition
  E_04 => next2 = next0 && temp1 = temp0
}}

fun Acyclic(s: S, x: List) {
  all e: x.(first.s).*(next.s) | e !in e.^(next.s)
}

fun specification() { with state {
  Acyclic(pre, l) && Acyclic(pre, m) => Acyclic(post, m)
}}

fun assertion() { encoding() && ! specification() }

run assertion for 2
```

Figure 3-16: Alloy model for the `swapTail` procedure - Part 2

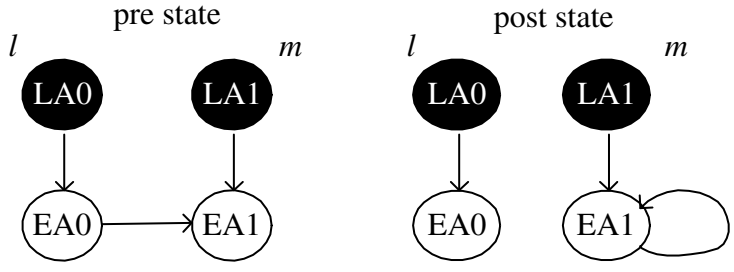

Figure 3-17: Counterexample for `swapTail` - revisited

Analyzer unwinds a universal quantifier internally into a conjunction.

The frame conditions required by the name replication encoding are small for field updates: they only concern the field being updated. At join points, this may not be the case, since they can involve many variables and fields. We minimize these, by not generating them for some variables whose values are not propagated beyond that point, making this kind of frame condition local to code blocks.

This is an improvement over frame conditions for the local-state idiom, where each update requires a condition on all other fields. The explicit-references encoding has generic frame conditions, but they involve a universal quantifier over references, which typically has a large scope, making the analysis less efficient.

Treatment of `null`

In the name replication encoding, `null` is represented by the empty set. This makes the implicit references idiom more expressive, because `null` is represented by a special `nullRef` signature instead, allowing sets and arrays to contain this value. We could represent `null` with a special atom for each type. But this increases the number of heap cells needed per class by one, making the analysis less efficient. We have therefore opted for a less expressive encoding, but a more efficient one. The practical consequence is that we can check bounded instances with higher scopes.

Tighter encoding in propositional logic

Our encoding has more relations than the common idioms, since each update produces a new instance for a field. This is not a problem because, as we will see in this section, it has nonetheless a tighter translation in propositional logic.

The next chapter will discuss how Alloy is translated to propositional logic. For the purposes of this section, it suffices to know that a relation of type $T \rightarrow V$ is represented by a matrix of $|T| \times |V|$ boolean variables, where $|T|$ and $|V|$ are the scopes of T and V, respectively, and each variable represents whether or not there is a mapping between corresponding atoms of T and V.

Consider a Java class C having k fields $f_1, \cdots, f_k$ of types T, and a procedure manipulating these. Assume that $|C| = |T| = n$. Each path of the computation graph for the procedure has a number of updates to some of these fields. Let u_i be the maximum number of updates to field f_i over all paths, and $u = \sum_{i=1}^{k} u_i$. Let s be the maximum number of updates over all paths. We have that $s \geq \mathrm{Max}_i u_i$, because s accounts for all updates to field f_j such that $u_j = \mathrm{Max}_i u_i$.

- In the local-state idiom, the fields become k relations of type:

$$C \rightarrow \texttt{State} \rightarrow T$$

  The number of `State` atoms required is the same as the maximum number of updates to any field over all paths. So $|\texttt{State}| = s$. The number of boolean variables required to represent a relation of this type in propositional logic is sn^2. So the total number of boolean variables over all the fields is: ksn^2.

Local-State	ksn^2	7350
Explicit Refs	$(4s + 2k)n^2$	9950
Name Replication	un^2	1800

Figure 3-18: Number of boolean variables needed, for a class C having k fields of type T, for scope n, where the maximum number of times f_i is updated over all paths is u_i, $u = \sum_{i=1}^{k} u_i$, and s is the maximum number of times any field is updated over all paths. On the right, actual numbers for insertion in red-black trees for 5 iterations and scope 5.

- The idiom with explicit object references has a relation `obj` of type:

  $$\texttt{State} \rightarrow \texttt{Ref} \rightarrow \texttt{Object}$$

  In addition, it has k fields of type:

  $$C \rightarrow \texttt{Ref}$$

  Since C and T must be subtypes of `Object`, its scope is $2n$. In the worst case, the number of references is the same as objects. The scope of `State` is s as above. So the number of boolean variables required for this scheme is: $4sn^2 + 2kn^2 = (4s + 2k)n^2$

- In our name replication encoding, field f_i is represented by u_i instances of type

  $$\texttt{state} \rightarrow C \rightarrow T$$

  Since `state` has scope 1, and is only used to encapsulate the state encoding, field f_i requires $u_i n^2$ boolean variables for all its instances. Summing this number over all fields yields: un^2.

Figure 3-18 summarizes the number of boolean variables required for each scheme. Since $s \geq \text{Max}_i u_i$, $ks \geq \sum_{i=1}^{k} u_i = u$. Therefore $ksn^2 \geq un^2$, and name replication requires fewer boolean variables than local-state. This is because the representation of each field in local-state grows larger with the total number of updates, including those to other fields. Thus update information is shared across the representation of fields.

On the other hand, name replication can require more variables than the explicit references scheme. This happens when many updates happen in parallel paths and s is much smaller than u. However, this is not the case in practice, and since s also grows larger with updates to variables that are not even fields, it is actually larger than u. For instance, for the insertion procedure in red-black trees presented

in Chapter 1, $k = 3$ (`left`, `right`, `parent`). For 5 iterations we have: $s = 98$, and $u = 72$. The right column of Figure 3-18 shows actual numbers for $n = 5$.

In conclusion, the small analysis of this section gives evidence that even though our scheme has more Alloy relations than the two idioms, it requires fewer boolean variables in practice, and has a tighter encoding at the level of propositional logic.

3.2 Extensions

In this section, we extend our basic encoding to handle different features of the Java language. Some features such as arrays, subclasses, and exceptions (except for detecting null pointer dereferences), are not currently implemented in our prototype tool, whose goal is to demonstrate the feasibility of the basic encoding and the optimizations for the common case of pointer manipulations. The sections on these features indicate that they can be easily supported in our framework. Of course, the tractability of the resulting analysis will remain to be seen. Other Java features such as multi-threading, reflection and dynamic loading, built-in exceptions other than null-pointer dereferences, are not supported and not discussed here.

3.2.1 Method Calls

In our technique, method calls are simply inlined. Recursion may be addressed by unwinding in a similar way to loops, but out prototype tool does not handle it.

To avoid shadowing variable names, each call gets a unique identifier (call id), and the local variables and parameters of the method are renamed using that identifier. Additional constraints are added to set the formals equal to the actual parameters. A special variable `result` holds the return value.

Constructors are inlined similarly, and the object created is held in a `result` variable. They additionally need the following constraints:

- The `result` variable is set equal to `this`.
- All class fields not mentioned in the body are set to be `null`.
- The returned object is chosen to be fresh. This is done by maintaining a special set, `used`, for each class, and constraining the new object not to be in this set.

Figure 3-19 shows a procedure that calls `swapTail`. Its computation graph is shown in Figure 3-20. Procedure `Main` invokes three methods: constructor for List (call id: 1), which also calls the constructor for `ListElem` (call id: 2), and `swapTail` (call id: 3). Local variables and parameters of methods and constructors are labeled with a call id, indicated with an underscore. For instance, `i1_1` refers to parameter `i` of the constructor for `List`.

The computation graph for the constructor of `ListElem` appears between nodes `1` and `6`. It starts out by setting the formal parameter equal to the actual, which in this case is `i1_1` passed on from the constructor of `List`. Then it sets the values of fields appropriately. Edge `E_45` is labeled with a constraint that states that the newly

```
class ListElem {
  int val;
  ListElem next;

  public ListElem(int i){
    val = i;
  }
}
class List {
  ListElem first;

  public List(int i){
    first = new ListElem(i);
  }

  static void swapTail(List l, List m){
    if (l.first != null
              && m.first != null) {
    ListElem temp = l.first.next;
    l.first.next = m.first.next;
    m.first.next = temp;
   }
  }

  void Main() {
    List l = new List(1);
    swapTail(l, l);
  }
}
```

Figure 3-19: Procedure that calls `swapTail`

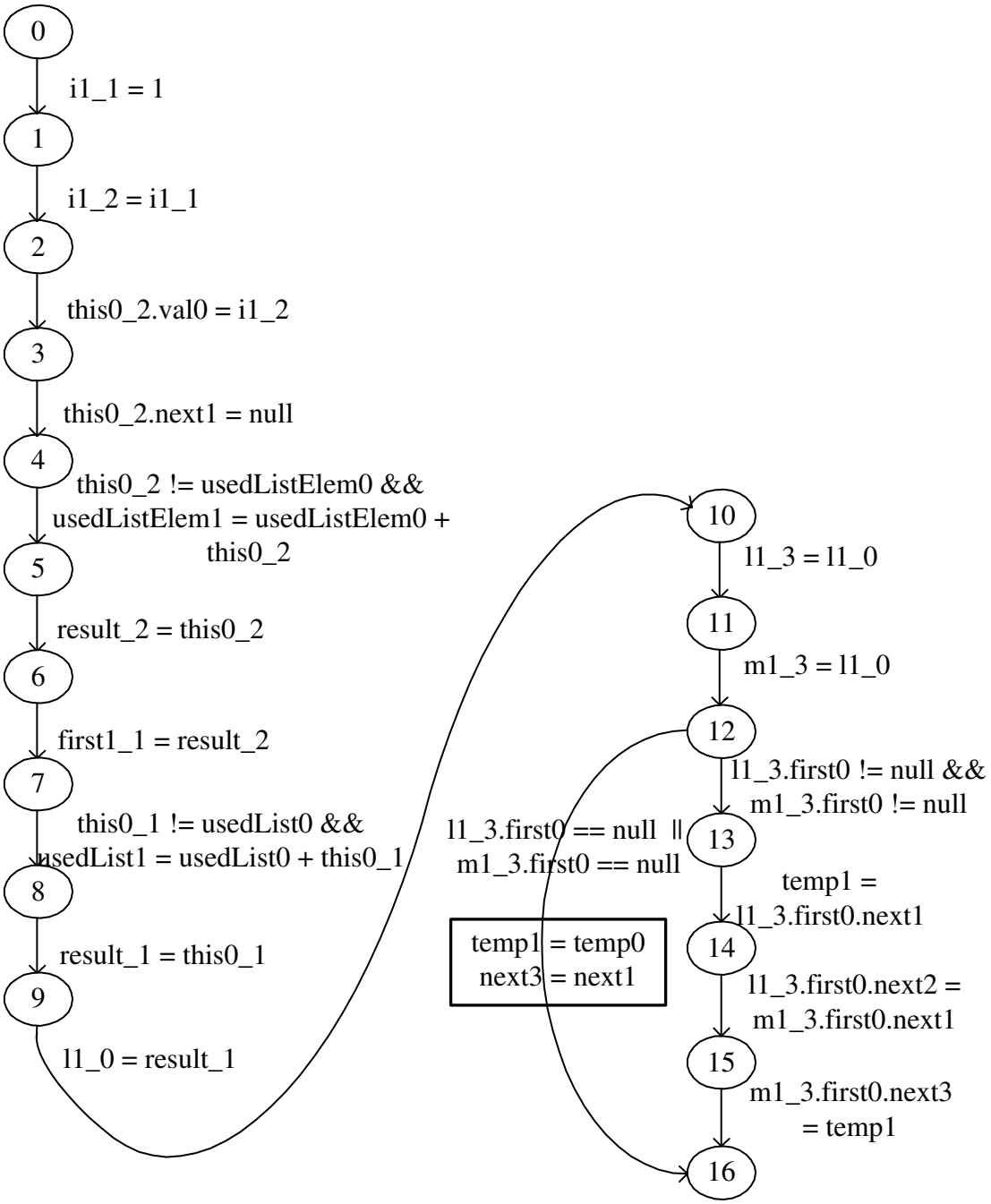

Figure 3-20: Computation graph for a procedure that calls `swapTail`

created object is fresh. Variable `usedListElem0` holds all the `ListElem` objects that are already in use in the heap. So this constraint simply says that `this0_2` is not in that set, and it updates the `used` set appropriately. Finally, edge `E_56` sets the result to be the same as `this0_2`. The computation graphs for the other calls are similar.

An edge label contains information about the file and the line number of the statement or condition that edge represents. This is used to convey the trace back to the user, when a counterexample is found.

3.2.2 Primitive Types

Our technique handles the primitive types `boolean` and `int`. As discussed before, we represent booleans with the type `Bit`, which has one atom. The truth value is then represented by testing for set-emptiness. So if `b` is a boolean, in Alloy, we write `some b` to indicate that it is true, and `no b` that it is false.

Java `int`s are represented by a signature `integer`. This allows for a very small number of `int`s – as many as the scope, but not necessarily in the range induced by the scope. We support comparisons between integers but not arithmetic. It is possible to incorporate a decision procedure for linear integer arithmetic in Alloy [30]. This will allow decoupling the scope from the number of integers that can be handled, and allow the analysis of arbitrary linear constraints over integers.

3.2.3 Arrays

Our technique has very limited support for integers. Its support for arrays is also therefore limited, since it cannot handle arithmetic for index manipulation. In this section we sketch how uni-dimensional arrays can be represented.
Arrays can be represented by Alloy functions ranging over non-negative integers:

- The length of the array is given by the cardinality of the function.
- An array initialization produces a constraint specifying the length of the array.
- Accessing the array is done by taking the relational image of the index.
- For each update of one of its elements, the array is renamed, and a frame condition is added to state the values at other indices does not change.

3.2.4 Subclasses

In this section, we sketch how subclasses can be represented, as well as inheritance and dynamic dispatch.

Java subclasses are represented in Alloy with subsignatures. Consider, for example, the Java program shown in Figure 3-21. It consists of a class `Point` and two subclasses `TwoDPoint` and `ColorPoint`, which inherit field `x`, and have additional fields `y` and `isRed`, respectively. All classes define a method `equals` which tests for equality by comparing respective fields.

```
public class Point {
  int x;
  public Point() {}
  public Point(int x){
    this.x = x;
  }
  public boolean equals(Point p){
    return (x == p.x);
  }
  public boolean transistiveTest(Point p1, Point p2, Point p3){
    boolean ret = false;
    if (p1.equals(p2) && p2.equals(p3) && p1.equals(p3))
      ret = true;
    return ret;
}}
public class TwoDPoint extends Point {
  int y;
  public TwoDPoint (int x, int y){
    this.x = x;
    this.y = y;
  }
  public boolean equals(Point p){
    if (x != p.x)
      return false;
    if (p instanceof TwoDPoint)
      return (y == ((TwoDPoint)p).y);
    return false;
}}
public class ColorPoint extends Point {
  boolean isRed;
  public ColorPoint (int x, boolean isRed){
    this.x = x;
    this.isRed = isRed;
  }
  public boolean equals(Point p){
    if (x != p.x)
      return false;
    if (p instanceof ColorPoint)
      return (isRed == ((ColorPoint)p).isRed);
    return false;
}}
```

Figure 3-21: Example of code with a class hierarchy

```
sig Point {
  x: integer
}
disj sig TwoDPoint extends Point {
  y: integer
}
disj sig ColorPoint extends Point {
  isRed: Bit
}
```

Figure 3-22: Example of Alloy representation of subclasses

Figure 3-22 shows the corresponding subtypes in Alloy. The signatures `TwoDPoint` and `ColorPoint` are declared to be disjoint, since the intersection of subclasses in Java is empty.

Field Inheritance

Field inheritance can be modelled naturally with subtypes in Alloy. For example, if `cp` is of type `ColorPoint`, then the field dereferencing expression `cp.x` is simply translated to the relational join expression `cp.x`. Since `cp` is in the set represented by signature `Point`, then `x` is well-defined on it.

Method Inheritance and Dynamic Dispatch

In Java, each expression has an *apparent* type as well as an *actual* one [23]. The apparent type is determined statically by the type checker based on type declarations. The actual type is the one an expression has at a program point when the program executes, and is possibly a subtype of the apparent one.

The procedure that is actually used in a method call depends dynamically on the actual type of the receiver (dynamic dispatch). It is the one declared in the class corresponding to the actual type, if it exists, and otherwise the one appearing in the nearest parent class (method inheritance).

We model method inheritance and dynamic dispatch in Alloy by explicitly testing for the type of the receiver and inlining all the possible computation graphs in parallel. Given a receiver with apparent type `T`, and a method `M` called on it, we select the declarations of `M` appearing in all the subtypes of `T`[3], as well as in the nearest parent. This gives a collection of computation graphs to be inlined. We then add a test for the type of the receiver to each graph and include them in an if-then-else construct, that does the tests starting from the subtypes that are farther away, to type `T` itself, and finishing with the parent. Since subtypes in Alloy are simply subsets, checking the type of an object is done in Alloy with set inclusion.

Figure 3-23 illustrates this by showing a fragment of the computation graph for procedure `Main` of Figure 3-21.

[3]We assume that a type is a subtype of itself.

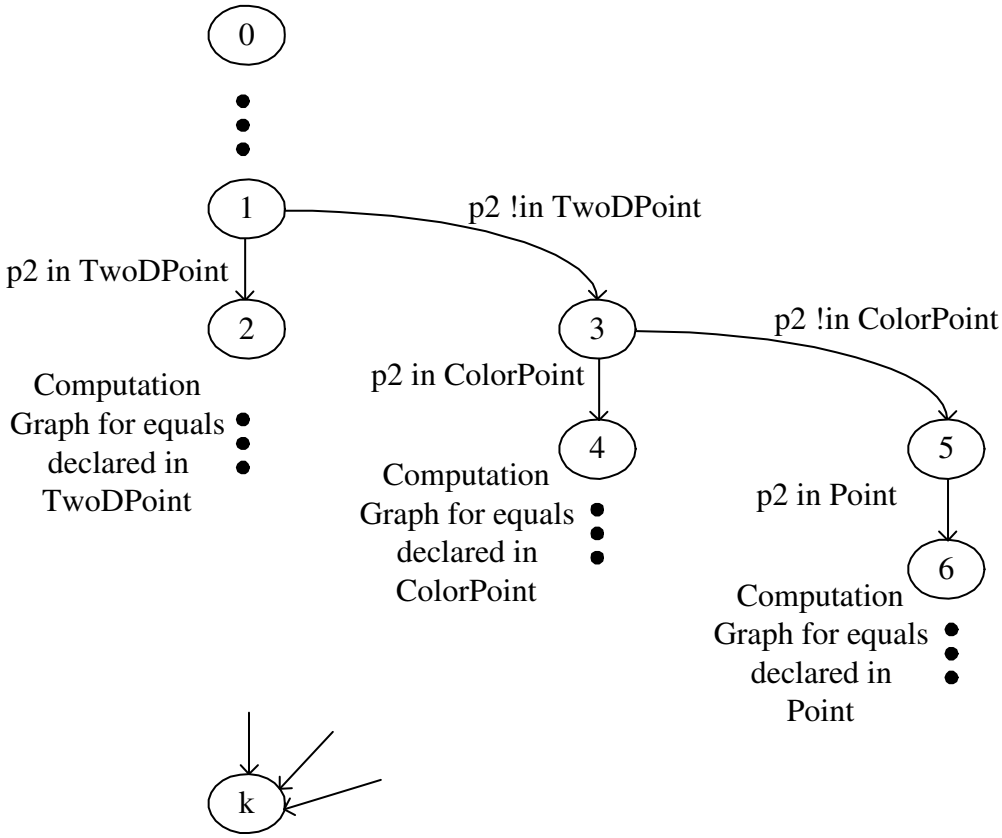

Figure 3-23: Computation graph for example with subclass hierarchy

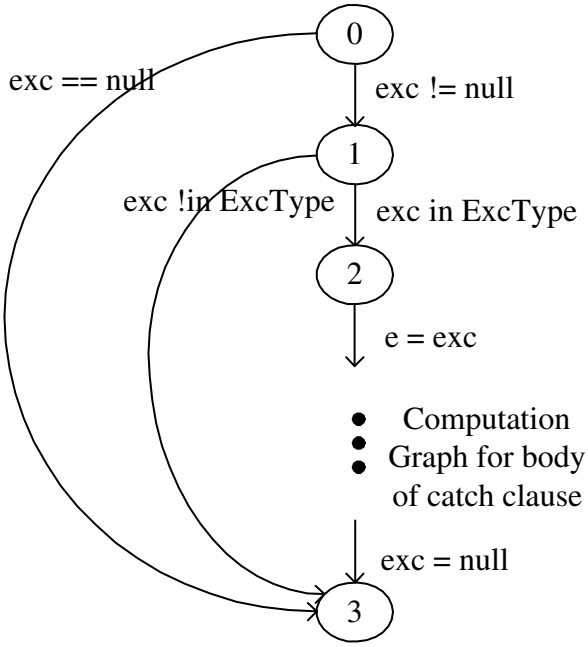

Figure 3-24: Computation graph for a `catch` block

3.2.5 Exceptions

To handle exceptions, we start with a computation graph whose method calls have been inlined. We use special variables to hold exception objects, and we modify the computation graph to accommodate exceptional control flow.

We introduce a signature `Exception` in Alloy and declare all exceptions to be its subtypes. The translation rules of Figure 3-11 are modified to include:

$$\mathcal{S}[\text{throw } e]\ i\ j \equiv \mathcal{E}[\text{exc}]\ j = \mathcal{E}[e]\ i$$

The variable `exc` represents the single live exception object, and has the same type. Edges labeled with throw statements are connected to the nearest enclosing catch clause, or to the exit point of the computation graph if none exists. The catch clause:

```
catch(ExcType e) { body }
```

gets the computation graph shown in Figure 3-24. First, there is a check to see if an exception was thrown by testing the variable `exc` against `null`. Since `null` is the empty set in our representation, this amounts to checking if `exc` is empty or not. The second condition checks if the exception object is of the appropriate type, and if so the body of the `catch` clause is executed, after setting the formal parameter equal to `exc`. Finally, the exception variable is reset to `null`. Multiple consecutive `catch` clauses get similar computation graphs connected together in series.

Detecting Null Pointer Dereferences

Null pointer dereference exceptions are handled in a similar way as user-defined exceptions. Consider the following fragment of code:

```
...
list = list.next;
...
```

We transform this code to the following:

```
...
if (list != null) list = list.next;
else throw new NullPointerException();
...
```

The `throw` statement is modeled in the computation graph as described above.

3.3 The Java API

In this section, we present how the Java API is handled in our analysis technique.

3.3.1 Overview

Instead of inlining method calls, we could instead replace them with an Alloy specification that describes their behavior. This avoids analyzing the body of the call, and makes the analysis scalable. However, it has the disadvantage of requiring the user to write such specifications, making the tool less practical. For the Java API, these specifications are known ahead of time. So our technique has a number of them built-in. In this section, we present the built-in specifications for `java.util.Set`, `java.util.Map`, and `java.util.Iterator`.

Each built-in specification consists of two parts: some declarations in the `state` sig, and a series of Alloy functions describing the behavior of specific methods. These functions take on the general form:

```
methodName(field, field': T -> U, ..., receiver: V, parameter: P,
                          ..., result: R)
```

The function takes all the fields that it modifies as parameters. This is because at each point when a call is made fields have been renamed, and the proper instances must be passed. The function also takes a receiver, the formal parameters of the corresponding method, and a result variable. The body of the function is a constraint relating all these parameters together.

Since subtypes are substitutable for their supertypes, they must obey the same specifications. We thus only give specifications for the interfaces.

3.3.2 java.util.Set

Figure 3-25 shows the specifications for `java.util.Set`. The `state` sig contains declarations for instances of field `elems`, which hold the elements of each `Set` in different states.

Following the `sig` declarations are specifications for various methods for `Sets`. The function `isEmpty` takes an `elems` field, a `Set s` and a return value `r`. The body of `isEmpty` says that if `s` has no elements then `r` is `true`, otherwise it is `false`.

The function `add` takes two instances of the `elems` field, for the `pre` and `post` states, and a receiver `Set s`. It also takes an object `o` of type `t`, and a return value `r`. The body says that the elements of `s` in the `post` state are the same as the ones for `s` augmented with object `o`. If `o` is not already in `s`, then `r` is `true`, otherwise it is `false`.

The function `iterator` returns a `java.util.Iterator` object, given a set `s`. In addition to the parameters `s`, return value `r`, and relevant fields, it also takes parameters `used` and `used'` of type `set Iterator`. These variables are needed whenever a new object is created, and indicate what objects are in use in each state. The function `iterator` takes these parameters since it creates a new `Iterator` object. Its body is described further in Section 3.3.3.

The specification for `java.util.HashSet` (Figure 3-26) contains only specifications for constructors and all other specifications are the same as those for `java.util.Set`.

It starts with a `sig` declaration, `HashSet`, to be a subtype of `Set`. The Java `HashSet()` constructor gets translated to the Alloy function `new`, which takes parameters `used`, `used'`, and return object `r` containing the newly created `HashSet`. The body of `new` simply says that `r` has no elements, and that it has not been previously used.

Note that we do not support `Sets` that contain `null` as an element, because `null` is represented as empty set.

3.3.3 java.util.Iterator

The `Iterator sig` is shown in Figure 3-27. It requires declarations of the fields `elems` and `seen`, for the set of elements for this iterator, and those that have already been returned, respectively. The body of the `iterator` function in the `Set` specification (Figure 3-25), says that the `Iterator r` has the same elements as `s`, and its `seen` set is empty. Moreover, `r` is a fresh object, it is not contained in `used`, but appears in `used'`.

The function specification for iterators are defined similarly to those for sets.

3.3.4 java.util.Map

The signature `Map` (`HashMap`) is shown in Figure 3-28 (Figure 3-29). It requires declarations of the field `map`, which hold the mapping of each `Map` object in each state. The function declarations for `Map` are similar to those for `Set`.

```
sig state {
...
elems0, elems1, ..., elemsk: Set -> Object,
...
}

sig Set {
  elems0, elems1, ..., elemsk: set Object
}

fun isEmpty(elems: Set -> Object, s: Set, r: boolean){
  no s.elems => some r, no r
}

fun contains(elems: Set -> Object, s: Set, o: Object, r: boolean){
  o in s.elems => some r, no r
}

fun add(elems, elems': Set -> Object, s: Set, o: Object, r: boolean){
   s.elems' = s.elems + o
   all e: Set - s | e.elems' = e.elems
   o !in s.elems => some r, no r
}

fun addAll(elems, elems': Set -> Object,
           s: Set, os: Set, r: boolean){
   s.elems' = s.elems + os.elems
   all e: Set - s | e.elems' = e.elems
   os.elems !in s.elems => some r, no r
}

fun remove(elems, elems': Set -> Object,
           s: Set, o: Object, r: boolean){
  s.elems' = s.elems - o
  all e: Set - s | e.elems' = e.elems
  o in s.elems => some r, no r
}

fun iterator(elems: Set -> Object, iteratorElems: Iterator -> Object,
             seen: Iterator -> Object, used, used': set Iterator,
             s: option Set, r: option Iterator){
  r.iteratorElems = s.elems
  no r.seen
  some r
  r !in used
  used' = used + r
}
```

Figure 3-25: Specification for java.util.Set

```
disj sig HashSet extends Set {}

fun new(elems: Set -> Object, used, used': set Set, r: HashSet){
  some r
  no r.elems
  r !in used
  used' = used + r
}
```

Figure 3-26: Specification for java.util.HashSet

```
sig state {
...
elems0, elems1, ..., elemsn: Iterator -> Object,
seen0, seen1, ..., seenm: Iterator -> Object,
...
}

sig Iterator {}

fun hasNext(elems: Iterator -> Object, seen: Iterator -> Object,
            i: option Iterator, r: option Bit){
  i.elems !in i.seen => some r, no r
}

fun next(elems, elems': Iterator -> Object,
         seen, seen': Iterator -> Object,
         i: option Iterator, r: option Object){
  r !in i.seen
  r in i.elems
  i.elems' = i.elems
  all e: Iterator - i | e.elems' = e.elems
  i.seen' = i.seen + r
  all e: Iterator - i | e.seen' = e.seen
}
```

Figure 3-27: Specification for java.util.Iterator

```
sig state {
...
map0, map1, ..., mapn: Map -> Object -> Object,
...
}

sig Map {}

fun isEmpty(map: Map -> Object -> Object,
            m: option Map, r: option Bit){
  no m.map => some r, no r
}
fun containsKey(map: Map -> Object -> Object,
                m: option Map,
                o: option Object, r: option Bit){
    some o.(m.map) => some r, no r
}
fun containsValue(map: Map -> Object -> Object,
                  m: option Map,
                  o: option Object, r: option Bit){
   some o.~(m.map) => some r, no r
}

fun get(map: Map -> Object -> Object, m: option Map,
        o: option Object, r: option Object){
   r = o.(m.map)
}
fun put(map, map': Map -> Object -> Object,
        m: option Map, o: option Object,
        v: option Object, r: option Object){
   m.map' = m.map ++ o -> v
   all e: Map - m | e.map' = e.map
   r = o.(m.map)
}
fun remove(map, map': Map -> Object -> Object,
           m: option Map, o: option Object,
           r: option Object){
   m.map' = m.map - o -> Object
   all e: Map - m | e.map' = e.map
   r = o.(m.map)
}
fun clear(map, map': Map -> Object ->Object, m: option Map){
  no m.map'
  all e: Map - m | e.map' = e.map
}
```

Figure 3-28: Specification for java.util.Map

```
disj sig HashMap extends Map {}

fun new(used, used': set Map, r: HashMap){
  no r.map
  r !in used
  used' = used + r
}
```

Figure 3-29: Specification for java.util.HashMap

Chapter 4

From Alloy to CNF

In this chapter, we discuss how the Alloy Analyzer translates Alloy to propositional logic (Section 4.1), and how propositional logic is then translated to conjunctive normal formal form, which is the input format of off-the-shelf SAT solvers (Section 4.2). This will serve as a preparation for the next chapter, which presents optimizations to improve scalability.

4.1 From Alloy to Propositional Logic

Alloy is a first-order relational logic. In order to translate it to propositional logic, the number of atoms per type must be bounded. This is given by the scope, which is supplied to our analysis technique by the user, as the number of heap cells.

Given a scope for each type, the Alloy Analyzer first allocates a matrix of boolean variables to each relation. Each of these boolean variables indicates whether a tuple is contained in the set of tuples for that relation.

An Alloy expression is translated into a matrix of boolean formulas. Each of these formulas is true if and only if the corresponding tuple is contained in the set of tuples of the relation denoted by that expression. These formulas are obtained by combining the matrices of sub-expressions following a given set of rules. To translate an Alloy formula to propositional logic, the Analyzer combines the matrices of its expressions, into a single formula. We present this translation more in detail in the following sections.

4.1.1 Allocating Boolean Variables

We illustrate with the translation of the Alloy formula:

$$a.f = b,$$

where a is a set of type A, b a set of type B, and f is a binary relation from A to B. For simplicity we assume that the scope of both A and B is n.

The Alloy Analyzer allocates a matrix of n^2 boolean variables to binary relation f:

$$\begin{matrix} f_{11} & \cdots & f_{1n} \\ \cdots & \cdots & \cdots \\ f_{n1} & \cdots & f_{nn} \end{matrix}$$

where f_{ij} is true if and only if f maps atom i of its domain to atom j of its range. Sets are degenerate relations. Set a is thus allocated n boolean variables:

$$\begin{matrix} a_1 \\ \cdots \\ a_n \end{matrix}$$

and similarly for set b:

$$\begin{matrix} b_1 \\ \cdots \\ b_n \end{matrix}$$

4.1.2 Translating Expressions

After having allocated boolean variables to all relations in the formula, the analyzer then proceeds to combine these matrices into matrices of boolean formulas. For example, given set a, the relational image $a.f$ of a under relation f, gets the matrix

$$\begin{matrix} (a_1 \wedge f_{11}) \vee \cdots \vee (a_n \wedge f_{n1}) \\ \cdots \\ (a_1 \wedge f_{1n}) \vee \cdots \vee (a_n \wedge f_{nn}) \end{matrix}$$

which is the result of matrix multiplication, and states that atom i is an element of $a.f$ if and only if there is some atom j in a such that f maps j to i. Other Alloy expressions are translated into matrices of boolean formulas in a similar way [14].

4.1.3 Translating Formulas

To translate formulas, the analyzer combines these matrices into a single propositional formula. For example, the formula $f = g$, where f and g are binary relations, becomes:

$$(f_{11} \Leftrightarrow g_{11}) \wedge \cdots \wedge (f_{1n} \Leftrightarrow g_{1n}) \wedge \cdots \wedge (f_{n1} \Leftrightarrow g_{n1}) \wedge \cdots \wedge (f_{nn} \Leftrightarrow g_{nn}).$$

which states that variables f and g denote the same set of tuples: tuple (i, j) is in f if and only if it is in g.

The formula $a.f = b$ then becomes the following conjunction:

$$\begin{gathered}((a_1 \wedge f_{11}) \vee \cdots \vee (a_n \wedge f_{n1})) \Leftrightarrow b_1 \wedge \\ \cdots \wedge \\ ((a_1 \wedge f_{1n}) \vee \cdots \vee (a_n \wedge f_{nn})) \Leftrightarrow b_n\end{gathered}$$

4.2 From Propositional Logic to CNF

Once a propositional formula is obtained, the Alloy Analyzer then proceeds to translate it to the input format of SAT solvers: conjunctive normal form (CNF), which consists of a conjunction of disjunctions of literals (i.e. a propositional variable, or its negation).

A simple way of doing this is to use distributivity laws. However, this results in an exponential blow-up in the size of the formula. Instead, the Alloy Analyzer uses a method due to to Plaisted and Greenbaum [32]. It consists of renaming each subformula – except negations and atomic propositions – with a fresh propositional variable. This causes an increase in the number of variables, but the size of the formula increases only linearly. In order to preserve the satisfiability of the original formula, a definition is conjoined for each new propositional variable. This definition is dependent upon the *polarity* of the subformula, defined below. Given a propositional formula ψ, we define in what follows $ren(\psi)$, the resulting formula after renaming, and then prove that it preserves satisfiability.

We write $\phi \sqsubseteq \psi$ to say that ϕ is a subformula of ψ. A satisfying assignment $\mathcal{S}$ to a formula is a set of pairs of propositional variables and truth values. We write $\mathcal{S}(\phi)$ to denote the truth value assigned to subformula ϕ by $\mathcal{S}$. The notation $\psi[V/\phi]$ means replacing subformula ϕ in ψ with V, where V is either a propositional variable or a truth value.

4.2.1 Polarity and Formula Renaming

When we rename a subformula ϕ with a propositional variable L, we need to conjoin a definition for L to the whole formula. This may be the following:

$$L \Leftrightarrow \phi$$

This guarantees that satisfiability is preserved, because L and ϕ are forced to have the same truth value. However, an equivalence ($\Leftrightarrow$) is expensive, since it results in two implications, and it is possible to simplify this definition into a one-way implication, given the notion of polarity. Informally, the polarity of a subformula indicates if there is an even or an odd number of negations leading to it, starting from the root of the syntax tree of the whole formula. The definition becomes an implication in one direction, or the other, depending on the polarity.

Polarity The polarity of a subformula ϕ of ψ, denoted by $pol(\phi,\psi)$, is defined as

follows. If there is an equivalence formula[1] ϕ', such that $\phi \sqsubseteq \phi' \wedge \phi' \sqsubseteq \psi$, then $pol(\phi,\psi) = 0$. Otherwise, if the number of negations plus the number of implications $\phi_1 \Rightarrow \phi_2$ such that $\phi \sqsubseteq \phi_1$ is even (odd), then $pol(\phi,\psi) = 1$ (-1).

Formula Renaming For each subformula ϕ in ψ to be renamed, we associate a literal L_ϕ, and a definition formula $Def(\phi)$ defined as follows depending on the polarity of ϕ relative to ψ:

$$Def(\phi) = \left| \begin{array}{l} L_\phi \Rightarrow \phi, \text{ if } pol(\phi,\psi) = 1 \\ L_\phi \Leftarrow \phi, \text{ if } pol(\phi,\psi) = \text{-1} \\ L_\phi \Leftrightarrow \phi, \text{ if } pol(\phi,\psi) = 0 \end{array} \right.$$

Let ψ' denote the formula ψ with all subformulas renamed to their corresponding literals. The formula $ren(\psi)$ is given by $\psi' \wedge \bigwedge_\phi Def(\phi')$.

Example Consider the formula $\neg(A \vee B) \vee (C \wedge D)$. Subformula $(A \vee B)$ has polarity 1, and $(C \wedge D)$ polarity -1. We can rename these with variables L_1 and L_2, respectively, and conjoin corresponding definitions, to obtain the renamed formula:

$$\begin{array}{c} L_1 \vee L_2 \wedge \\ L_1 \Leftarrow (A \vee B) \wedge \\ L_2 \Rightarrow (C \wedge D) \end{array}$$

A subformula of polarity 1 has the property that, if a satisfying assignment for the whole formula assigns the truth value *false* to it, then that value is really a "don't care". This means that if it is replaced with *true*, then the whole formula is still satisfied.

Conversely, a subformula of polarity -1 has a similar property: if a satisfying assignment assigns the *true* to it, then that value is really a "don't care".

In this example, consider the satisfying assignment that assigns *false* to both $(A \vee B)$ and $(C \wedge D)$. Then it is also a satisfying assignment for $\neg(A \vee B) \vee true$. This explains why the one-way implication in $L_2 \Rightarrow (C \wedge D)$ is adequate: if $(C \wedge D)$ is *true* then L_2 must also be true, but if is false, then L_2 can be either *true* or *false*, and the whole formula is still satisfied.

Property Let ϕ be a subformula of ψ with polarity 1 (-1). If $\mathcal{S}$ is a satisfying assignment such that $\mathcal{S}(\phi) =$ false (true), then $\mathcal{S}$ is also a satisfying assignment for $\psi[true/\phi]$ $(\psi[false/\phi])$.

Theorem A formula ψ is satisfiable if and only if $ren(\psi)$ is satisfiable.

[1] An equivalence formula is one where the top-most node of its syntax tree is an equivalence.

Proof We show that renaming a single subformula ϕ preserves satisfiability. $ren(\psi)$ can then be obtained by a series of renamings. So we prove that ψ is satisfiable if and only if $\psi[L_\phi/\phi] \wedge Def(\phi)$ is satisfiable.

Assume ψ is satisfiable and let $\mathcal{S}$ be a satisfying assignment. Then $\mathcal{S} \cup \{(L_\phi, \mathcal{S}(\phi))\}$ is a satisfying assignment for $\psi[L_\phi/\phi] \wedge Def(\phi)$ regardless of the polarity of ϕ.

Now assume that $\psi[L_\phi/\phi] \wedge Def(\phi)$ is satisfiable and let $\mathcal{S}$ be a satisfying assignment. We show that ψ is satisfiable as well, by cases:

- $pol(\phi,\psi) = 1$, $Def(\phi) = L_\phi \Rightarrow \phi$. If $\mathcal{S}$ assigns the same truth value to L_ϕ and ϕ, then $\mathcal{S}$ is also a satisfying assignment for ψ. If $\mathcal{S}$ assigns false to L_ϕ and true to ϕ. In the formula $\psi[L_\phi/\phi]$, L_ϕ has polarity 1, so by the Property above, $\mathcal{S}$ is also a satisfying assignment for $\psi[L_\phi/\phi][\text{true}/L_\phi]$. But under the assignment $\mathcal{S}$, $\psi[L_\phi/\phi][\text{true}/L_\phi] = \psi$. So $\mathcal{S}$ is necessarily also a satisfying assignment for ψ.

- $pol(\phi,\psi) = -1$, $Def(\phi) = L_\phi \Leftarrow \phi$. Analogous to the case above.

- $pol(\phi,\psi) = 0$, $Def(\phi) = L_\phi \Leftrightarrow \phi$. In this case, $\mathcal{S}$ must assign the same truth value to ϕ and L_ϕ, so it is also a satisfying assignment for ψ.

4.2.2 Example

Consider the subformula

$$(A \wedge B) \vee (C \wedge D) \vee (E \wedge F) \vee (G \wedge H) \vee (I \wedge J) \vee (K \wedge L).$$

If we use distributivity laws to transform this to CNF, we obtain 64 clauses:

$$\begin{gathered}
(A \vee C \vee E \vee G \vee I \vee K) \\
(A \vee C \vee E \vee G \vee I \vee L) \\
(A \vee C \vee E \vee G \vee J \vee K) \\
(A \vee C \vee E \vee G \vee J \vee L) \\
(A \vee C \vee E \vee H \vee I \vee K) \\
(A \vee C \vee E \vee H \vee I \vee L) \\
(A \vee C \vee E \vee H \vee J \vee K) \\
(A \vee C \vee F \vee H \vee J \vee L) \\
(A \vee C \vee F \vee G \vee I \vee K) \\
(A \vee C \vee F \vee G \vee I \vee L) \\
\cdots
\end{gathered}$$

Using Plaisted and Greenbaum's method, we first rename the subformulas, and obtain the following conjunction of formulas:

$$
\begin{gathered}
L_1 \vee L_2 \vee L_3 \vee L_4 \vee L_5 \vee L_6 \\
L_1 \Rightarrow (A \wedge B) \\
L_2 \Rightarrow (C \wedge D) \\
L_3 \Rightarrow (E \wedge F) \\
L_4 \Rightarrow (G \wedge H) \\
L_5 \Rightarrow (I \wedge J) \\
L_6 \Rightarrow (K \wedge L)
\end{gathered}
$$

Each of the lines above can be easily translated to CNF:

$$
\begin{gathered}
(L_1 \vee L_2 \vee L_3 \vee L_4 \vee L_5 \vee L_6) \\
(\neg L_1 \vee A) \\
(\neg L_1 \vee B) \\
(\neg L_2 \vee C) \\
(\neg L_2 \vee D) \\
(\neg L_3 \vee E) \\
(\neg L_3 \vee F) \\
(\neg L_4 \vee G) \\
(\neg L_4 \vee H) \\
(\neg L_5 \vee I) \\
(\neg L_5 \vee J) \\
(\neg L_6 \vee K) \\
(\neg L_6 \vee L)
\end{gathered}
$$

This has 13 clauses, instead of 64. In general, there is a linear increase in the size of the formula, as opposed to an exponential one.

Chapter 5

Optimizations

A field declared in a class is modelled as a relation, but it always maps its object to exactly one other object (or to null). In other words, the relation is *functional*. Mathematically, this relation is a function. By exploiting this fact, we can optimize different steps of the analysis, with the goal of reducing the number of variables and clauses produced in the final CNF, and thus improving the SAT solver's performance.

Our main idea is a compact representation for functions that requires fewer boolean variables than the representation for general relations. Alone, however, this does not reduce the number of variables in the CNF, because the step that translates propositional formulas to CNF adds intermediate variables, and counteracts the benefit of the improved representation. To obtain a real benefit, we need two other optimizations: first, a systematic introduction of variables in the first-order formula, and, second, a series of logical simplifications in the propositional formula. In the next sections, we describe how these optimizations work independently, and how they fit. But first, we give a justification for why the size of the CNF is a good metric and why it is useful to aim at reducing it.

5.1 Reducing CNF Size

It is generally hard to predict the efficiency of SAT solvers, and the best way to validate an optimization is through empirical results. Indeed, the efficiency of a SAT solver is dependent on the *structure* of the formula being analyzed. A large formula may take a short time because the SAT solver can find a small inconsistent subset of the clauses quickly, or deduce an inconsistency rapidly through resolution. A short formula may take a long time because its structure may be more intricate.

A smaller CNF size does not always mean that the SAT solver will run faster. Indeed there is a competition [34] for the smallest hard CNF. Moreover adding clauses sometimes may improve efficiency. An example of that is one of Alloy's optimizations, symmetry-breaking predicates [36], that improve performance, while adding more clauses. These work by restricting the structure of the formula, allowing the SAT solver to converge faster.

However, there is a limit in terms of number of clauses and variables that state-

of-the-art SAT solvers can handle. Beyond that limit they take too long or take too much memory regardless of the structure of the formula. Every boolean variable may potentially double the run-time, since the SAT solver searches for a solution with that variable having alternatively value true and false. It is therefore sensible to reduce the number of variables. Moreover, 80% of the SAT solver's time is spent doing unit propagation[1], which consists of going through the clauses and setting their literals to true or false, after having made a decision on the value of one variable. This time increases monotonically with the number of clauses. It is therefore also sensible to reduce the number of clauses.

In the next sections we present semantics-preserving optimizations that allow a tighter encoding of code in CNF both in terms of the number of variables and clauses. The next chapter provides empirical evidence that they result in an improvement in the run-time, and allow formulas previously unanalyzable to be analyzed.

5.2 Function Representation

A relation f that is a total function maps each atom in its domain to exactly one atom in its range. By representing this atom as an integer in binary form, the encoding of f requires only $\lfloor log(n) \rfloor + 1$ rather than n boolean variables in each row. Suppose the tighter encoding of f is the matrix:

$$\begin{array}{ccc} f_{11} & \cdots & f_{1l} \\ \cdots & \cdots & \cdots \\ f_{n1} & \cdots & f_{nl} \end{array}$$

Not all relations in our encoding are functions. For instance, the transitive closure of a field (which may appear in a specification) is not generally functional. Therefore, the relational and functional representation must coexist. To achieve this, we transform the tighter representation to the standard, $n \times n$, by building the $n \times n$ matrix from the $n \times (\lfloor log(n) \rfloor + 1)$ variables of the smaller matrix:

$$\begin{array}{ccc} \neg f_{11} \wedge \cdots \wedge \neg f_{1(l-1)} \wedge f_{1l} & \neg f_{11} \wedge \cdots \wedge f_{1(l-1)} \wedge \neg f_{1l} & \cdots \\ \cdots & \cdots & \cdots \\ \neg f_{n1} \wedge \cdots \wedge \neg f_{n(l-1)} \wedge f_{nl} & \neg f_{n1} \wedge \cdots \wedge f_{n(l-1)} \wedge \neg f_{nl} & \cdots \end{array}$$

where the formula at row i and column j is true if and only if row i in the compact representation of f represents integer j. Note that since $\lfloor log(n) \rfloor + 1$ bits can represent more than n values, we must add a side condition that constrains each row of the compact representation in such way that it represents an integer less than n.

If we incorporate this optimization in the Alloy Analyzer, this actually results in an *increase* in the number of variables in the final CNF. This is because the step that transforms propositional logic to CNF counteracts the gain of the compact representation, by renaming all the formulas in the converted matrix with propositional

[1]This is according to the developers of Chaff [27], a state-of-the-art SAT solver.

variables (as described above in Section 4.2). So the resulting CNF has all the variables that it would have had without the compact representation, in addition to all the ones the new representation introduces.

5.3 Introducing Alloy Variables

To avoid this problem, we first rename all subexpressions that are scalars, i.e. singleton sets, in the first-order formula. This breaks formulas into small subformulas that are translated in a non-compositional fashion. Most subformulas that appear in the translation of a fragment of Java code have the form:

$$v.f_1.\cdots.f_{k_1} = u.g_1.\cdots.g_{k_2},$$

where v and u are scalars, and $f_1, \cdots, f_{k_1}$, and $g_1, \cdots, g_{k_2}$ are functions encoding fields. We rename subexpressions of the form $a.f$ by introducing an Alloy variable b, and conjoin definitions of the form $b = a.f$ with the whole formula. Variable b is a scalar since a is a scalar and f is a function. We obtain:

$$v_1 = v.f_1 \wedge \cdots \wedge v_{k_1-1} = v_{k_1-2}.f_{k_1-1} \wedge u_1 = u.g_1 \wedge \cdots \wedge u_{k_2} = u_{k_2-1}.g_{k_2} \\ \wedge\ v_{k_1-1}.f_{k_1} = u_{k_2}.$$

The next section describes logical simplifications that allow a formula of the form $a.f = b$ to be translated compactly to CNF without adding any additional propositional variables. The CNF's for these formulas are then conjoined by taking the union of their clauses. Introducing an Alloy variable for the subexpression $a.f$ results in $\lfloor log(n) \rfloor + 1$ additional boolean variables. If this subexpression were translated to CNF without the introduction of Alloy variables and logical simplifications, it would result in at least n^2 additional boolean variables, since all the elements of the product (an $n \times n$ matrix) would be renamed.

5.4 Logical Simplifications

We now describe the logical simplifications that allow us to translate $a.f = b$ compactly to CNF, without introducing any additional propositional variables. They take advantage of the fact that a scalar is represented by a collection of propositional formulas having the property that exactly one of them is true. Informally, the first two simplifications help because they push disjunctions down in the formula's syntax tree. Disjunctions are a source of blow-up when transforming to CNF, and their effect is lessened if they are further away from the root.

Logical Simplification 1 Consider the formula:

$$(A_1 \wedge B_1) \vee \cdots \vee (A_n \wedge B_n) \tag{5.1}$$

where the A_i and B_i $(1 \leq i \leq n)$ are boolean formulas. If exactly one of the A formulas is true, then it can be easily seen that (5.1) is logically equivalent to:

$$(\neg A_1 \vee B_1) \wedge \cdots \wedge (\neg A_n \vee B_n) \tag{5.2}$$

Logical Simplification 2 Consider the formula:

$$((A_1 \wedge B_1) \vee \cdots \vee (A_n \wedge B_n)) \Leftrightarrow C \tag{5.3}$$

where A_i and B_i $(1 \leq i \leq n)$ are boolean formulas. If exactly one of the A formulas is true, then it can be easily seen that (5.3) is logically equivalent to:

$$(A_1 \wedge (B_1 \Leftrightarrow C)) \vee \cdots \vee (A_n \wedge (B_n \Leftrightarrow C)) \tag{5.4}$$

Our final simplification is specific to the representation of integers, and relies on the fact that integers can be compared bit by bit.

Definitions A *literal* is either a propositional variable, or its negation of one. Given a literal a, let $var(a)$ denote the propositional variable corresponding to a, and $phase(a)$ be $+$ $(-)$ if a is $var(a)$ $(\neg var(a))$.

Logical Simplification 3 Let A_i $(1 \leq i \leq n)$ be a collection of formulas of the form $a_1^i \wedge \cdots \wedge a_l^i$, such that for all i, j, and for all k $(1 \leq k \leq l)$, $var(a_k^i) = var(a_k^j)$, and let B_i be a similar collection. Consider the formula:

$$A_1 \Leftrightarrow B_1 \wedge \cdots \wedge A_n \Leftrightarrow B_n \tag{5.5}$$

If exactly one of the A_i is true, and similarly for the B_i, and for all i and k, $phase(a_k^i) = phase(b_k^i)$, then it can be seen that (5.5) is logically equivalent to:

$$var(a_1^1) \Leftrightarrow var(b_1^1) \wedge \cdots \wedge var(a_l^1) \Leftrightarrow var(b_l^1) \tag{5.6}$$

5.5 Using the Optimizations

Let us now see how these optimizations are applied in obtaining the CNF for $v.f_1.\cdots.f_{k_1} = u.g_1.\cdots.g_{k_2}$.

We have seen that $v.f_1.\cdots.f_{k_1} = u.g_1.\cdots.g_{k_2}$ can be transformed into a conjunction of $k_1 + k_2$ formulas of the form $a.f = b$. Let k denote $k_1 + k_2$. Since conjunction of CNF can be obtained simply by taking union of clause sets, we can avoid variable introduction and obtain a formula of size $k\alpha$, if $a.f = b$ can be represented with α clauses.

In what follows, we compute α. Consider translating the formula $a.f = b$ to CNF. The variable a is a scalar and its compact representation requires l boolean variables: $a_1 \cdots a_l$, where l denotes $\lfloor log(n) \rfloor + 1$, and n is the scope. Function f has a compact representation having nl boolean variables: $f_{11} \cdots f_{1l} \cdots f_{n1} \cdots f_{nl}$.

Converting the compact representation of a to the standard one results in a vector of n elements. We use A_i to denote the formula on row i of this vector, and similarly B_i for b. Function f's compact representation results in an $n \times n$ matrix, and we use F_{ij} to denote the formula at row i, column j.
The formula $a.f = b$ can be translated as:

$$\begin{gathered}(A_1 \wedge F_{11} \vee \cdots \vee A_n \wedge F_{n1}) \Leftrightarrow B_1 \\ \wedge \cdots \wedge \\ (A_1 \wedge F_{1n} \vee \cdots \vee A_n \wedge F_{nn}) \Leftrightarrow B_n\end{gathered}$$

Exactly one of the A_i is true, so we can apply Logical Simplification 2:

$$\begin{gathered}A_1 \wedge (F_{11} \Leftrightarrow B_1) \vee \cdots \vee A_n \wedge (F_{n1} \Leftrightarrow B_1) \\ \wedge \cdots \wedge \\ A_1 \wedge (F_{1n} \Leftrightarrow B_n) \vee \cdots \vee A_n \wedge (F_{nn} \Leftrightarrow B_n)\end{gathered}$$

We can then apply Logical Simplification 1:

$$\begin{gathered}(\neg A_1 \vee (F_{11} \Leftrightarrow B_1)) \wedge \cdots \wedge (\neg A_n \vee (F_{n1} \Leftrightarrow B_1)) \\ \wedge \cdots \wedge \\ (\neg A_1 \vee (F_{1n} \Leftrightarrow B_n)) \wedge \cdots \wedge (\neg A_n \vee (F_{nn} \Leftrightarrow B_n))\end{gathered}$$

After moving terms around and factoring, we obtain:

$$\begin{gathered}\neg A_1 \vee ((F_{11} \Leftrightarrow B_1) \wedge \cdots \wedge (F_{1n} \Leftrightarrow B_n)) \\ \wedge \cdots \wedge \\ \neg A_n \vee ((F_{n1} \Leftrightarrow B_1) \wedge \cdots \wedge (F_{nn} \Leftrightarrow B_n))\end{gathered}$$

Note that for all i, the formulas $F_{i1}, \cdots, F_{in}$ and $B_1, \cdots, B_n$ satisfy the conditions of Logical Simplification 3. So we apply it to obtain:

$$\begin{gathered}(\neg A_1 \vee ((f_{11} \Leftrightarrow b_1) \wedge \cdots \wedge (f_{1l} \Leftrightarrow b_l))) \\ \wedge \cdots \wedge \\ (\neg A_n \vee ((f_{n1} \Leftrightarrow b_1) \wedge \cdots \wedge (f_{nl} \Leftrightarrow b_l)))\end{gathered}$$

The intuition behind this final formula is the following. If A_i is true, this means that the i^{th} atom is included in a, and since a is a scalar, that is its value. In that case, the i^{th} row of the compact representation of f – which contains the value mapped to the i^{th} atom – must be equal point-wise to that of b, since $a.f = b$.

This formula can be easily translated to CNF, since e.g. $f_{11} \Leftrightarrow b_1$ is $(f_{11} \Rightarrow b_1) \wedge (f_{11} \Leftarrow b_1)$, and the A_i are conjunctions, whose negations become disjunctions, and can be easily distributed. Therefore, formula $a.f = b$ results in $2nl$ clauses, and no additional intermediate variables. The formula $v.f_1.\cdots.f_{k_1} = u.g_1.\cdots.g_m$ results in $2nlk$ clauses, and since we added k variables to break it down, it has lk intermediate boolean variables.

	Clauses	Intermediate Variables
Non-Optimized	$(4k+2)n^2 + kn$	$kn^2 + kn$
Optimized	$2kn\lfloor log(n) \rfloor + k$	$k\lfloor log(n) \rfloor + k$

Figure 5-1: Number of clauses and intermediate variables for $v.f_1.\cdots.f_{k_1} = u.g_1.\cdots.g_{k_2}$ for scope n

Consider translating $v.f_1.\cdots.f_{k_1} = u.g_1.\cdots.g_{k_2}$ to CNF, without using optimizations. Each subexpression of the form $a.f$ results in a vector of n formulas, that are disjunctions of n conjunctions. For each of these formulas, we introduce n propositional variables to rename the conjunctions, requiring 3 clauses each for their definitions. We also introduce 1 variable to rename the whole formula, and its definition requires $n+1$ clauses. So the subexpression $a.f$ requires $n(n+1)$ additional variables, and $n(4n+1)$ clauses. Therefore there are $n(4n+1)k$ clauses and $n(n+1)k$ variables after the translation of each side of the equality. The equality itself adds $2n^2$ clauses. We obtain the numbers summarized in Figure 5-1. Some typical numbers (see next chapter) are 58056 clauses in the optimized case vs 153259, and 13384 variables instead of 64575. These numbers were taken from the analysis of a procedure manipulating red-black trees.

Chapter 6

Case Studies

In this chapter, we present a series of case studies to demonstrate the effectiveness of our optimizations, and the usefulness of our technique. These case studies were performed with a prototype tool, Jalloy, that implements our technique for the basic subset of Java presented in Section 3.1. Jalloy has built-in specifications for the Java API given in Section 3.3. It works by translating Java code directly to CNF using the optimizations, using Alloy to translate the specifications to CNF, and conjoining the two. All experiments were run on a 1.1GHz PentiumIII with 640MB of memory, using the BerkMin SAT solver [10].

6.1 Red-black Trees

We visit again the red-black tree example presented in the introduction. The goal of this case-study is to demonstrate the efficacy of the optimizations in practice.

6.1.1 Code

Recall red-black trees are binary search trees whose nodes have an additional attribute, a color, which is either red or black. Each procedure manipulating red-black trees must preserve some invariants regarding colors, that together maintain trees that are roughly balanced.

Figures 6-1 and 6-2 reproduce an implementation of insertion in a red-black tree. This code is a close transcription of pseudocode presented in a popular algorithms book [6]. It contains two classes `RBNode` and `RBTree`, and procedure `RBInsert`, which performs insertion.

The comment `/@verify@/` indicates to the prototype tool which procedure to check. A more sophisticated prototype could support writing specifications directly in code as annotations, and would avoid the need for this comment. Our tool takes the input parameters, the number of iterations and bound on the size of the heap, on the command line.

```
class RBNode {
  boolean isRed; int key;
  RBNode right; RBNode left;
  RBNode parent;
  public RBNode(int i){
    isRed = false; key = i;
  }
}
class RBTree {
  RBNode root;
  void TreeInsert(RBNode z){
    RBNode k = null;
    RBNode x = this.root;
    while (x != null){
      k = x;
      if (z.key < x.key) x = x.left;
      else x = x.right;}
    z.parent = k;
    if (k == null) this.root = z;
    else if (z.key < k.key) k.left = z;
    else k.right = z;
 }
 void LeftRotate(RBNode z){
   RBNode y = z.right;
   z.right = y.left;
   if (y.left != null)
     y.left.parent = z;
   y.parent = z.parent;
   if (z.parent == null)
     this.root = y;
   else if (z == z.parent.left)
     z.parent.left = y;
   else
     z.parent.right = y;
   y.left = z;
   z.parent = y;
 }
 void RightRotate(RBNode z){
   // similar to LeftRotate, left and right swapped
 }
```

Figure 6-1: Left and right rotation for red-black trees

```
// /@verify@/
void RBInsert(int i){
  RBNode h = new RBNode(i);
  this.TreeInsert(h);
  h.isRed = true;
  while (h != this.root && h.parent.isRed == true){
    if (h.parent == h.parent.parent.left){
      RBNode y = h.parent.parent.right;
      if (y != null && y.isRed == true){
        h.parent.isRed = false;
        y.isRed = false;
        h.parent.parent.isRed = true;
        h = h.parent.parent;
      } else {
        if (h == h.parent.right) {
          h = h.parent;
          this.LeftRotate(h);
        }
        //h.parent.isRed = false; //bug seeded
        h.parent.parent.isRed = true;
        this.RightRotate(h.parent.parent);
      }
    } else { //same as above with
             // left and right inverted
    }
    this.root.isRed = false;
  }
 }
}
```

Figure 6-2: Code for insertion in red-black trees

6.1.2 Specification

Each procedure manipulating red-black trees must preserve the following invariants:

1. If a node is red, then both of its children are black.
2. All paths from the root to a node with at most one child have the same number of black nodes.

In order to check these invariants, the user writes a specification file (Figure 6-3). Functions `CorrectColors` and `IsBalanced` express invariants 1 and 2 respectively. Function `Tree` states some well-formedness conditions on a red-black tree: the parent field must be acyclic, and left, right and parent fields must be set consistently.

The variables available to the user for writing specifications are the parameters of the procedure to be checked and `This` if it is non-static. These variables are declared in the `state` signature, and are made available to the user with the `with state` construct.

Most functions typically take a parameter of type `S` that is an enumerated type containing elements `pre` and `post`. These are used to indicate the state of a field. For example, `n.(left.pre)` denotes the object pointed to by the `left` field of `n` in the `pre` state.

6.1.3 Results

The results are shown in Figure 6-5. Times are in seconds.

Some experiments were done after injecting a bug by removing one line in the code (indicated in Figure 6-2 by the comment `bug seeded`). In Figure 6-5, dashes indicate either that the experiment took more than 10 minutes or that there was a shortage of memory. The non-optimized experiments are done by translating a subset of Java code to Alloy and uses the Alloy Analyzer equipped with BerkMin as well. The number of iterations are generally picked to match the scope, since that will allow a structure of that size to be traversed.

Some of these experiments result in a counterexample. For instance, the counterexample corresponding to assertion `CCAssertBis` with the bug seeded, for scope of 5 and 5 iterations, is reproduced in Figure 6-4. The numbers on each node indicate the keys, and red nodes are shown in white. The counterexamples goes through 5 iterations to violate `CCAssertBis`.

The results show a considerable reduction in the number of clauses and variables for the optimized case, a significant improvement in run-time, and the ability to check larger instances.

They also show that the translation without optimization can obtain all the counterexamples very rapidly. This was expected; a fundamental assumption of our work, which we refer to as the *small scope hypothesis*, is that most bugs can be demonstrated with small counterexamples. An empirical study [1] demonstrates that a scope of 6 is enough to obtain full statement and branch coverage for a variety of benchmarks. Our optimizations allow checking all properties with a scope of 6, which was not possible before, and as high as scope of 8 in some cases.

```
fun Tree(s: S) { with state {
  no (This.s).(root.s).(parent.s)
  all r : (t.s).(root.s).*(left.s + right.s) {
    r !in r.^(parent.s)
    some r.(parent.s) => r in r.(parent.s).(right.s + left.s)
    some r.(left.s + right.s) => r.(left.s) != r.(right.s)
    some r.(right.s) => r.(right.s).(parent.s) = r
    some r.(left.s) => r.(left.s).(parent.s) = r
  }
 }
}
fun CorrectColors(s: S) { with state {
  all r: (This.s).(root.s).*(left.s + right.s) |
    some r.(isRed.s) => no r.(right.s).(isRed.s) &&
                        no r.(left.s).(isRed.s)
 }
}
fun CCAssert() { with state {
  Tree(pre) && CorrectColors(pre) => CorrectColors(post)
 }
}
fun CCAssertBis() { with state {
  Tree(pre) && CorrectColors(pre)
    && IsBalanced(pre) => CorrectColors(post)
 }
}
fun HasAtMostOneChild(r: RBNode, s: S){
  no r.(right.s) || no r.(left.s))
}
fun IsBalanced(s: S) { with state {
  all r1, r2: (This.s).(root.s).*(left.s + right.s) {
    HasAtOneMostOneChild(r1, s) &&
    HasAtOneMostOneChild(r2, s) =>
       #{r: RBNode | r in r1.*(parent.s) && no r.(isRed.s)}
      =
       #{r: RBNode | r in r2.*(parent.s) && no r.(isRed.s)}
  }
 }
}
fun IBAssert() {
  Tree(pre) && IsBalanced(pre) => IsBalanced(post)
}

fun specification() { CCAssert() }
```

Figure 6-3: Specification for red-black tree insertion

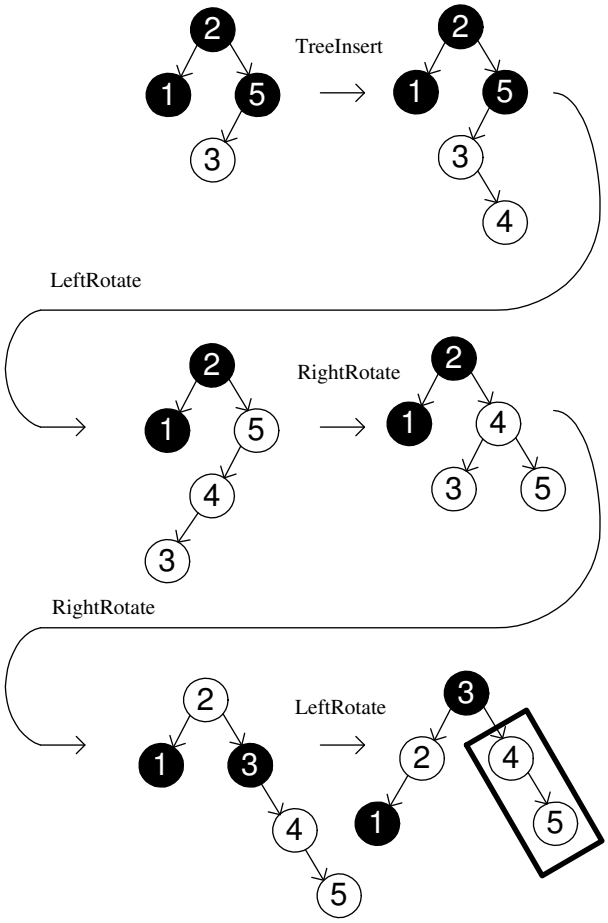

Figure 6-4: Counterexample for Insertion in Red-Black trees - revisited

We can also increase the number of iterations to 20 and get an outcome within a minute or two in most cases. For `RBInsert`, 20 iterations for each loop correspond to 1540 lines of code. A state-of-the-art SAT solver can handle formulas with up to about 300,000 clauses. These results suggest that a code fragment of 1500 lines might be encodable within these limits. Of course, the tractability of the subsequent analysis is another matter, and depends on the intricacy of the structure of the code.

assertion	scope	# iter	counter?	time opt	clauses opt	vars opt	time	clauses	vars
CCAssert	2	2	no	0	6827	2218	0	15343	7638
	3	3	no	0	14443	4201	0	40042	18727
	4	4	no	3	35642	7887	20	82678	36316
	4	10	no	4	79496	15033	82	188452	81148
	4	20	no	13	152586	26943	–	–	–
	5	5	no	22	58056	13384	232	153259	64575
	5	10	no	43	102281	19854	506	283780	117344
	5	20	no	25	190731	32794	–	–	–
	6	6	no	159	85160	19659	–	–	–
	6	10	no	198	126700	25247	–	–	–
	6	20	no	514	230550	39217	–	–	–
IBAssert	2	2	no	0	7066	2310	0	15545	7694
	3	3	no	0	16300	4816	0	41827	19281
	4	4	yes	7	40950	9559	15	87654	37766
	4	10	yes	25	84804	16705	87	193404	82574
	4	20	yes	109	157894	28615	–	–	–
	5	5	yes	12	87369	22109	52	182058	72853
	5	10	yes	44	131594	28579	–	–	–
	5	20	yes	144	220044	41519	–	–	–
	6	6	yes	22	148548	38027	103	318908	121400
	6	10	yes	49	190088	43615	–	–	–
	6	20	yes	132	293938	57585	–	–	–
	7	7	yes	141	240865	62127	–	–	–
	7	10	yes	189	276652	66627	–	–	–
CCAssertBis bug seeded	2	2	no	0	6867	2242	0	15537	7726
	3	3	no	0	15165	4471	0	41172	19162
	4	4	no	2	37886	8679	18	85840	37450
	4	10	no	12	81302	15753	82	192286	82630
	4	20	no	62	153662	27543	–	–	–
	5	5	yes	18	75141	18608	65	172059	70344
	5	10	yes	43	118921	25013	140	303460	123573
	5	20	yes	100	206481	37823	–	–	–
	6	6	yes	77	124936	31383	202	298686	116224
	6	10	yes	82	166056	36915	–	–	–
	7	7	yes	93	199567	50690	–	–	–
	7	10	yes	239	234991	55145	–	–	–
	8	8	yes	272	349823	82336	–	–	–
	8	10	yes	641	386261	86216	–	–	–

Figure 6-5: Results for red-black tree insertion

6.2 Garbage Collection

In this section, we present an implementation of the Shorr-Waite garbage collection algorithm. This was posed as a Challenge Problem at the 2001 Microsoft Summer Institute[1].

6.2.1 Code

The `garbageCollector` procedure takes a `root` node and a set of nodes constituting a graph. It marks all nodes that are reachable from the root, collects all the unmarked ones in a free list, and returns the first element of this list.

6.2.2 Specification

There are two correctness requirements:

1. The left and right fields of a node reachable from the root are not changed.

2. Any node on the free list is not reachable from the root.

Figure 6-7 shows these specifications. The helper function `reachable` defines what it means for a node to be reachable from the root. Function `reachablesUnchanged` states the first property above, and `freeUnreachable` the second one.

6.2.3 Results

Running Jalloy for a scope of 4 and 4 iterations on both properties results in no counterexample. We now seed a bug in the code (indicated by the comment `bug seeded`), and remove the part of the `mark` procedure that visits the right field of nodes.

We obtain the counterexample for the `reachablesUnchanged` property in 14 seconds (Figure 6-8). It shows two nodes `N0` and `N2`. `N0`'s right field points to `N2`, and `N2`'s to itself. Left fields are null. Since right fields are not explored in the buggy code, `N2` does not get marked. It thus becomes the first element of `freeList` and its right field gets set to `null`, violating the property that reachable nodes must not change.

This case study illustrates the fact that Jalloy is easy to use, and its specification language expressive enough to handle properties of interest. By ease of use, we mean that no annotations beyond the property of interest were needed for the analysis.

6.3 Using Jalloy to Debug Jalloy

In this section, we present a case study where Jalloy was used to find a real bug, unknown to the author, in the code of Jalloy itself. The bug was found in the following

[1]The code we consider here is written using loops and not recursion.

```
class Node {
  boolean marked;
  Node left;
  Node right;
}
import java.util.*;
class GC {
  // /@verify@/
  public Node garbageCollector(Node root, Set nodes) {
    Iterator t = nodes.iterator();
    while(t.hasNext()){
      Node node = (Node) t.next();
      node.marked = false;
    }

    mark(root);

    Node freeList = null;
    Iterator r = nodes.iterator();
    while(r.hasNext()){
      Node node = (Node) r.next();
      if (node.marked == false){
        node.left = freeList;
        node.right = null;
        freeList = node;
      }
    }
    return freeList;
  }

  // Marks all nodes that a reachable from "from".
  private void mark(Node from) {
    Set work = new HashSet();
      work.add(from);
      while(!work.isEmpty()){
        Iterator t = work.iterator();
        Node n = (Node) t.next();
        work.remove(n);
        if (!n.marked){
          n.marked = true;
          if (n.left != null)
            work.add(n.left);
          //if (n.right != null)  //bug seeded
          //  work.add(n.right);
        }
      }
   }
}
```

Figure 6-6: Code for the garbage collection algorithm

```
fun reachable(n: Node){with state {
  n in (root.pre).*(left.pre + right.pre)
 }
}

fun reachablesUnchanged() {with state {
  all n: Node | reachable(n) => {
n.(left.pre) = n.(left.post)
n.(right.pre) = n.(right.post)
  }
 }
}

fun freeUnreachable(){with state {
  all n: Node | n in (freeList.post).*(left.post) <=> !reachable(n)
 }
}

fun specification(){
  reachablesUnchanged()
}
```

Figure 6-7: Specification for the garbage collection procedure

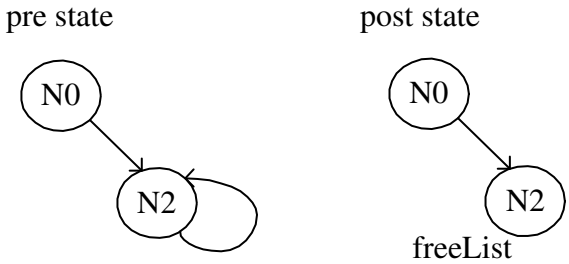

Figure 6-8: Counterexample for assertion `reachablesUnchanged`

way. We first observed that some procedure did not have the required behavior. However the calling context was too large for Jalloy to handle. We thus analyzed the procedure against its preconditions and specification, to see if the problem came from the procedure itself. Jalloy did not find a counterexample, which indicated that the procedure was correct, and probably one of its preconditions was violated in the code. By relaxing preconditions one by one, we observed different scenarios in which the procedure broke, and that helped to pinpoint which precondition was not satisfied.

Our technique can be used in this fashion to get assurance that a procedure is correct, but also to gain a better understanding of how a violations of its preconditions affect behavior. Therefore the tool supports reasoning about code, a procedure at a time.

6.3.1 Code

The code appears in Figures 6-9 and 6-10[2]. Computation graphs are represented with sets of edges. Each edge has a set of incoming edges, `InEdges`, and outgoing ones, `outEdges`. Computation graphs are DAGs with a single entry and exit nodes. We do not represent nodes explicitly. The starting edges are all those having no incoming ones, and the final edges are those having no outgoing ones. The method `getFirstEdges` returns takes a set of edges and returns those having no incoming ones. Method `getNextEdges` also takes a set of edges, and returns the set of all of their outgoing ones.

The method `topologicalSort` returns the topological sort of a set of edges that represent a computation graph. It has a working set, `current`, which is initialized to the initial edges of the graph. In each iteration, the elements of `current` are added, in any order, to the vector `ord`, `current` is updated to be the next set of edges, and the new `current` set is removed from `ord`. These next edges are then added back in the next iteration to the end of the vector, thus respecting the partial order.

6.3.2 Specification

During regular testing, we observe that `topologicalSort` returns a vector that does not have every edge appearing in the `edges` input parameter. So we write the specification of Figure 6-11 to check the following property:

- The vector returned, `ord`, has the same elements as the input parameter `edges`.

The specification makes use of field `elems` in the specification of `java.util.Set`, which represents the set of objects contains in the collection.

The `precondition` function expresses some preconditions and well-formedness constraints. Formula `1` says that `outEdges` is acyclic, and similarly for `inEdges`. Formula `3` states that `inEdges` and `outEdges` must be consistent, and Formula `4` that if edge `e` is in `edges`, then so are its incoming and outgoing edges.

[2]In the experiment, we replaced the Vector type by Set, since Vectors are not supported by the prototype tool. This is adequate for the specification we consider.

```
import java.util.*;
public class Edge {
  Set inEdges;
  Set outEdges;

  public Edge(){
    inEdges = new HashSet();
    outEdges = new HashSet();
  }

  public boolean hasNoInEdges(){
    return inEdges.isEmpty();
  }

  public static Set getFirstEdges(Set edges){
    Set firstEdges = new HashSet();
    Iterator t = edges.iterator();
    while(t.hasNext()){
      Edge e = (Edge) t.next();
      if (e.hasNoInEdges())
        firstEdges.add(e);
    }
    return firstEdges;
  }

  public static Set getNextEdges(Set edges){
    Set nextEdges = new HashSet();
    Iterator t = edges.iterator();
    while(t.hasNext()){
      Edge e = (Edge) t.next();
      nextEdges.addAll(e.outEdges);
    }
    return nextEdges;
  }
}
```

Figure 6-9: Code for Jalloy - Part 1

```
public class ComputationGraph {
  Set edges;

  ///@verify@/
  public Vector topologicalSort(){
    Vector ord = new HashSet();
    Set current = Edge.getFirstEdges(edges);
    while(!current.isEmpty()){
      ord.addAll(current);
      current = Edge.getNextEdges(current);
      ord.removeAll(current);
    }
    return ord;
  }
}
```

Figure 6-10: Code for Jalloy - Part 2

The `correct` is the correctness condition: `ord` and `edges` must have the same set of elements. The assertion checks to see if the precondition implies the correctness condition.

6.3.3 Results

We check procedure `topologicalSort` for 2 iterations and a scope of 5, and obtain no counterexample. If we comment out precondition `3`, we obtain the counterexample shown in Figure 6-12, where circles represent edge atoms, light arrows are `inEdges` fields, and dark ones `outEdges`. The returned vector contains atom `E4`.

The `current` set is initialized to contain `E4` since this has no incoming edges. Because of the inconsistency between `inEdges` and `outEdges`, atom `E3` is never added to `current`, and therefore is missing from the output vector.

Going back to the context in which `topologicalSort` is called, we discover that precondition `3` is indeed violated. The following helped in finding this bug:

- Getting confirmation that the procedure was right in itself.
- Writing the preconditions down.
- Relaxing preconditions to examine how the procedure breaks.
- Checking the calling code to see if preconditions are indeed violated.

```
fun precondition() {with state {
  -- 1. outEdges are acyclic
  Acyclic((outEdges.pre).(elems.pre))

  -- 2. inEdges are acyclic
  Acyclic((inEdges.pre).(elems.pre))

  -- 3. inEdges and outEdges are consistent
  all e1, e2: Edge | e2 in e1.(outEdges.pre).(elems.pre) <=>
                              e1 in e2.(inEdges.pre).(elems.pre)

  -- 4. If e is in edges, then so are its inEdges and outEdges
  all e: Edge |
    e in (This.pre).(edges.pre).(elems.pre) =>
         e.(outEdges.pre).(elems.pre)
                   in (This.pre).(edges.pre).(elems.pre) &&
         e.(inEdges.pre).(elems.pre)
                   in (This.pre).(edges.pre).(elems.pre)
}

fun correct() {with state {
  (ord.post).(elems.post) = (This.pre).(edges.post).(elems.post)
 }
}

fun assertion() { precondition() => correct() }

fun specification() { assertion() }

fun Acyclic [t] (r: t -> t) {no x: t | x in x.^r}
```

Figure 6-11: Specification for Jalloy

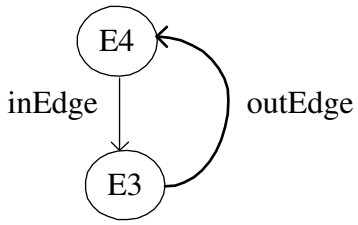

Figure 6-12: Counterexample Jalloy

Chapter 7

Conclusion

In this chapter, we first discuss how our analysis technique fits in the larger context of a software engineering process. We then present related work. The chapter concludes with an evaluation of the merits and deficiencies of the thesis work, and final thoughts.

7.1 Jalloy in Context

Software bugs may be roughly categorized as follows. One type of error arises when a program is solving the wrong problem. A software system is often part of a larger system consisting of hardware, specialized devices, and in some cases humans. Specifying the precise requirements of a software system is difficult and error-prone in itself. Software may therefore meet its requirements, but fail to fulfill its purpose. Another category of errors is when a program does not satisfy its high-level requirements, and these arise for several reasons.

- **Algorithmic Bugs**. The system implements an algorithm correctly but the algorithm does not solve the problem. Or else the system implements an algorithm incorrectly. This category includes bugs that result from typos in the code.
- **Bugs of Omission**. The system does not handle all possible inputs correctly. There is basically missing code, resulting from unforeseen usages of the system, or from wrong assumptions. This category also includes bugs in exceptional termination, which arise when the system does not handle failure gracefully.
- **Bugs of Resource Management**. The system may take too much memory or too much time, failing to meet its performance requirements. It may have memory leaks because it fails to release memory adequately, or there are unexpected buffer overflows.
- **Security Bugs**. The system satisfies its functional requirements when used within the envelope it was built for, but is vulnerable to attacks by malevolent clients.

Our analysis technique is independent of any model or methodology for software development. It does not address testing systems as a whole, or showing that a system meets its high-level requirements. It is useful for checking aspects of an individual module, specifically abstract data types with rich complex structure, during implementation.

The technique does not require a whole program and can be used on incomplete code. It ensures that a procedure is algorithmically correct and provides the right functionality, by checking that if its preconditions are met then so are its postconditions. It can also be used to show that preconditions are satisfied in a calling context. Moreover, the user may check that a data representation is manipulated correctly by a procedure, by showing that key invariants are maintained.

In addition to uncovering algorithmic bugs, the technique can also unveil bugs of omission. A test suite with good code coverage exercises existing code, but may not uncover bugs that arise from missing code. Our analysis technique can uncover these types of bugs because we are checking consistency between code and a specification. Any discrepancy, whether from existing or missing code, is reported as an error. It may therefore find subtle bugs that a good test suite misses.

The technique can also be used to find bugs of resource management. By setting small bounds on the lengths of buffers, the analysis can be used to check for buffer overflows. Security bugs are often a result of buffer overflows, so these types of errors may be unveiled as well.

Our analysis also complements existing testing techniques in that it supports informal reasoning about the code. When a test case fails, it may not be clear where the bug is located, and performing deep analysis of small fragments of the code can help to reason about it, and uncover the source of the error. Suppose, for example, that a programmer observes that the output of a called procedure is not what he expects. Two scenarios are possible: either the procedure is incorrect, or its preconditions are violated in the calling context. By analyzing the procedure itself he can disambiguate between these two scenarios. If it appears to be correct, then its preconditions are violated in the calling context, and the error is located elsewhere.

Our analysis also allows the programmer to gain a better understanding of how violations of preconditions affect the behavior of a procedure. By weakening preconditions, he may use our technique to generate scenarios in which the procedure fails. This in turn can be useful for reasoning about the calling context. Moreover, the analysis can help to pinpoint preconditions and postconditions when these are not known by the programmer. These can be used as documentation for the code, but more importantly can suggest new test cases.

7.2 Related Work

For a bounded instance of a program, our analysis explores *all* the possible inputs and executions, typically accounting for billions of cases. Unlike testing, it can also generate an initial configuration of the heap which leads to a property violation. It differs from finite state verification tools, such as model checking, in that it is modular:

procedures may be checked in isolation without requiring a driver. It differs from shape analysis in that it produces sound counterexamples and no false alarms. It also requires no intermediate code annotations, or user-provided abstractions.

7.2.1 TestEra

TestEra [25] is a specification-based testing method for Java. It consists of generating test cases from an Alloy specification of a procedure. The precondition is used to produce all the non-isomorphic inputs of a bounded size, which are concretized to Java objects. TestEra then runs the code on these concrete inputs, to produce concrete outputs, which are mapped to abstract ones. The abstract outputs is checked for consistency against the Alloy postcondition. Any discrepancy is reported as an input/output pair that violates the specification. TestEra has been used successfully to test methods from the Java Collection API, and a fault tree analyzer. It has helped to find bugs in real software systems.

TestEra can be seen as a dynamic version of our technique. The code is not modeled in Alloy, but executed instead. This poses a tradeoff. On the one hand, it scales better than our technique. On the other hand, the number of test cases produced can grow very large and it may be prohibitive to execute the code for all of them. Our technique does not actually execute different paths of the code, but finds a single execution that violates a property. It has therefore an advantage for specifications that produce a large number of possible heap shapes (e.g. unconstrained graphs).

7.2.2 Finite State Verification

FeaVer

Telephone switches offer over 100 features, and the distributed software required for call processing is very complicated. The interaction between these features is the main source of complexity: 10 of them alone produce 2^{10} possible combinations, and testing is simply infeasible. This is known as the *feature interaction problem*. FeaVer [13] is a tool that addresses this problem. It has been used to find hundreds of bugs in Lucent's PathStar access server.

FeaVer supports mechanized model extraction, to translate C code to Promela, the input language of the Spin model checker [12]. The user is required to provide a lookup table that maps patterns in the code to Promela. Three different kinds of abstraction are possible: disregarding a statement and replacing it with `skip`, preserving a statement entirely, or mapping to a non-deterministic construct. Non-determinism is used to abstract irrelevant data, and to model the environment of the system in a general way. The lookup table needs to be constructed once. As the system evolves, it changes only slightly.

FeaVer uses the lookup-table to produce an abstract model of a C program, which it then verifies against a user-provided temporal specification, using the model checker Spin. It outputs any counterexample trace found back to the user in a suitable fashion.

FeaVer differs from our technique in that it does not target structural properties of code. Moreover, it requires user-provided abstraction which our technique does not need.

Bandera

Model checking techniques have matured to the point that they are used for hardware verification in industry. The transfer of technology for software has been difficult because of a gap between modern programming languages and the input languages of model checkers (typically finite state machines). The goal of the Bandera project [5] is to address this gap, by providing automated support for model construction.

Bandera consists of a slicer that disregards components of the system not relevant to the property being checked, an abstraction-based specializer that helps the user to limit the domain of variables and to perform abstract interpretation, and a back end that maps Bandera's intermediate representation to the input languages of various existing model checks (such as Spin [12] and SMV [26]).

Like FeaVer, Bandera is used to check temporal properties of code and is not targeted at the kind of structural properties we consider. It also requires user-provided abstractions.

Java PathFinder

The Java PathFinder [41] is an environment for checking Java bytecode, that integrates model checking, program analysis, and testing. It requires user-provided abstractions of the program, and uses the Bandera tool for slicing. Java PathFinder requires an initialization of the heap that fixes it to a particular configuration. Thus it is impossible to have the tool automatically find an initial configuration that breaks an assertion, as can be done in our analysis.

A recent approach [21] to verifying Java programs is a generalization of symbolic execution to linked data structures. It is based on the Java PathFinder and augments it with the capability of symbolic execution of Java code.

The tool takes a fragment of Java code and a property to verify. It then proceeds to symbolically execute it: variables of primitive types are given a symbolic value, and fields of objects are given the value *unknown*. These fields are given actual values in a process called *lazy initialization*: when a field needs to be accessed, it is initialized in all the possible ways, producing a tree of executions visited in turn. The branching conditions for each execution are kept in a *path condition*. When these conditions involve linear integer arithmetic, a decision procedure (the Omega library [31]) is used to determine if they are consistent. When the path condition for an execution becomes inconsistent, the tool does not pursue it any further. A counterexample is output to the user when a property is found not to be satisfied.

This approach essentially consists of executing all possible paths and treating only primitive types as symbolic values. It is based on model checking, which suffers from the state explosion problem. It is not clear what the memory requirements are for this method, when checking a procedure with complex pointer manipulations. To

ease this problem, the approach makes use of preconditions, to disallow configurations that do not satisfy them. For example, if a list is known to be acyclic, fewer heap configurations are possible, and therefore not all executions are explored.

Another drawback of the approach is that it is non-terminating for structural properties that are satisfied. For example, consider a while loop that traverses a list, breaking only when it reaches null. If no counterexample is found, this approach will continue to explore longer and longer lists, without any provisions for when to stop. For such properties, running the approach for long enough gives a guarantee that the property is satisfied. But there is no evidence as to what that length of execution is for a given program.

Our approach is different in that it always terminates, regardless of the satisfiability of the property to be checked. Moreover, our technique does not explore all the possible executions. These are encoded in the logic and the constraint solver finds an execution violating the property in a goal-oriented way, by trying to satisfy a logical formula. Our approach is limited by the capability of the SAT solver, but does not suffer as badly from state explosion, because it does not need to maintain information about all the possible states.

SLAM

The SLAM [2] toolkit takes a C program and a safety property and checks fully automatically if the property is satisfied. It outputs either a sound 'yes' or 'no', or 'don't know'. It can prove that a property is satisfied, and if it is not, it outputs a counterexample trace, with no spurious error reports. Because of the incompleteness of the underlying theorem prover, the answer may also be 'don't know'. Since verifying properties is undecidable, SLAM may not always terminate. It has however been shown to terminate in practice.

The toolkit is composed of three main parts: C2BP performs an abstraction of the C program based on *predicate abstraction*; BEBOP is a novel model checker that uses a combination of data-flow analysis and binary decision diagrams (BDDs); NEWTON is a predicate discovery tool.

The process works as follows. The user writes a specification in SLIC (Specification Language for Interface Checking), modelling a state machine expressing erroneous executions. The C code is then instrumented with the specification, and the problem becomes: is the error state reachable? The C2BP tool takes an instrumented program and abstracts it into a *boolean program*, in which the only variables are booleans and represent predicates of the original program. BEBOP model checks the boolean program to determine if the error state is reachable. If it is not, there is a guarantee that the property is satisfied because boolean programs are sound abstractions. If the error state is found to reachable, then NEWTON determines if the counterexample found is feasible. If it is, a sound counterexample has been found. If it not, NEWTON produces a set of new predicates that explicate the infeasibility. These predicates result in new boolean variables, and a new boolean program is built, at which point the process starts over.

Predicate abstraction was first introduced by Saidi and Graf [11] in the context of

verifying infinite-state systems. It consists of mapping the concrete states to abstract states according to their evaluation under a finite set of predicates. The abstract transitions are computed using a theorem prover. C2BP realizes predicate abstraction for a substantial subset of C. It starts with an initial set of predicates corresponding to the control conditions of the C program. It assigns a boolean variable for each of these predicates. To determine how the value of the booleans variables is affected with each statement, it uses weakest preconditions [8]. Since the weakest precondition itself may not be an existing predicate, it must be strengthened to a combination of existing ones. This is done by using decision procedures of existing theorem provers (Simplify and Vampyre), and potentially an exponential number of calls to the theorem prover must be made (in the number of predicates). A number of optimizations alleviate this issue in C2BP. The tool also uses alias analysis to improve the precision of the weakest precondition computation in the presence of pointers and aliasing.

The BEBOP tool uses data-flow analysis to compute the set of reachable states for each statement in a boolean program. It uses BDDs to represent sets of states, as well as the transfer function of each statement. Unlike symbolic model checkers, it does not represent the control-flow with a BDD, but keeps an explicit representation of the control flow graph. It computes a fixpoint by iterating over the set of facts associated with each statement.

SLAM does not require any user-annotations or user-provided abstractions. Instead, it performs its own abstraction and automatically refines it. It has been used successfully to find bugs in device drivers, and checking API usage rules. The process converges rapidly for control-intensive properties, but not necessarily for data-intensive ones – the kind of structural property we consider for our technique. SLIC is more limited than our technique's specification language in terms of what it can express, since transitive closure is not available.

7.2.3 Shape Analysis

Shape analysis algorithms can identify invariants for programs that manipulate heap-allocated storage. They represent the heap as *shape graphs*, conservative abstractions that capture properties at different points in the program.

Parametric shape analysis (PSA) [33] is a technique that represents the heap as a structure in a 3-valued logic. Variables are unary predicates, and fields binary ones. The boolean value 1/2 is used to represent uncertainty, and is used to abstract sets of concrete heaps. Like traditional shape analysis, there are "summary nodes" that represent one or more concrete heap cells, and are represented by a special unary predicate.

PSA starts with a set of possible input heap shapes, represented as 3-valued logical structures, and performs abstract interpretation over them, given the semantics of each statement in the program. The process terminates because the structures considered are bounded. It computes the shape of the heap at each program point. The invariants discovered can then be used to see if some property of interest is satisfied. The output of the analysis is a sound 'no', or 'yes', and when nothing can be deduced 'don't know'.

To make the analysis more precise, PSA allows the user to provide *instrumentation predicates*. An example of such a predicate is: do two or more fields point to the same element? They are associated with a definition in terms of the core predicates, but store more precise information than evaluating that formula. They are used to fine tune the analysis.

The ideas in PSA have been implemented in a tool, TVLA [22], that takes as input the initial shape graphs, as well as the operational semantics of each statement in the code's control-flow graph, together with instrumentation predicates. It has been used on code manipulating linked data structures, as well as sorting programs.

Unlike our technique, PSA can prove properties without bounds, but it may also output 'don't know'. In contrast to our property-independent translation, PSA requires instrumentation predicates that tailor the analysis for the discovery of particular properties. Recent work [24] presents space-efficient encodings of boolean formulas that represent shape graphs. Its goals but not its methods are similar to ours.

7.2.4 Theorem Proving

ESC

The Extended Static Checker (ESC) [7] uses a specialized theorem prover to check code. It can detect null-pointer dereferences, array out-of-bound errors, and can also check user-defined pre and post-conditions. These specifications can define abstract variables that respect hiding: the internal variables of a procedure are not exposed. A mechanism is provided for expressing abstraction functions that connect abstract variables to their representation.

ESC translates the code together with user annotations to Dijkstra's guarded commands, and uses weakest preconditions to generate verification conditions. These are then handed to a theorem prover, Simplify, which runs automatically with no user assistance. ESC is not geared at proving correctness, but rather finding bugs. In particular, failed proofs are turned into useful error messages for the user.

There are several aspects of the design of ESC that make it unsound. The user may write `Assume` statements to prevent spurious error messages, but the correctness of these constraints is up to the user. Experience with ESC has shown that unwinding loops up to a limited number (0 or 1) leads to the discovery of many useful errors, and is more efficient than discovering loop invariants. This is another source of unsoundness by design.

Structural properties such as those handled in our technique are not expressible in ESC's annotation language, mainly because of the absence of transitive closure. Since we consider bounded programs only, transitive closure over a relation becomes the application of that relation as many times as the scope. So given any scope, our properties involving transitive closure are expressible in ESC's annotation language, by repeated field dereferencing. However, this notation is cumbersome, and any change in the scope entails a change in the annotations. Experiments have not been performed to compare the tools where ESC is used in this fashion.

ESC outputs error messages when a property is found to be violated, but unlike

our technique, no counterexample trace.

Verifun

Verifun [9] is a theorem proving system that grew out of experience with the Simplify prover, and is based on the conventional Nelson-Oppen design [28]. It leverages recent advances in SAT solving technology to perform backtracking more efficiently.

Many verification problems can be expressed in a logical formula containing predicates and functions from an application domain composed together with logical connectives. Verifun separates the propositional aspect of the formula by introducing proxy variables. These are propositional variables that are introduced to replace predicates and functions, and therefore abstract away their specific semantics. The abstract propositional formula obtained is conservative: it has a representation of all the satisfying assignments of the original formula, and possibly more.

Verifun then uses an off-the-shelf SAT solver to obtain a solution. If none is found, then there is a guarantee that the original formula did not have a solution. On the other hand, if there is a solution, the tool must check that it is not spurious, by verifying that the truth values of the proxy variables are consistent with the underlying semantics of the predicates they represent. This is done by invoking decision procedures for, e.g. linear arithmetic, and the theory of arrays.

These decision procedures are augmented with an explicating capability. When a set of constraints are found to be unsatisfiable, a proof is generated in terms of the relations between the proxy variables. This additional constraint is then conjoined with the abstract propositional formula and the process starts over. Thus, this approach performs automated successive refinement of an abstract formula, and adds more and more detailed information about the underlying semantics of the predicates that are abstracted away.

The idea of replacing an entity with a boolean variable is similar to predicate abstraction, where predicates are first introduced to capture some aspect of a system, such as its control flow information. Then the predicates are replaced by boolean variables similar to the proxy variables in Verifun, and analysis proceeds given this abstraction. SLAM's boolean programs are abstractions of this nature.

Verifun does not handle the kind of structural property that we consider in our technique.

7.3 Evaluation

In this section, we evaluate the merits and deficiencies of the thesis, as well as opportunities for future work and challenges that lie ahead in order to make the tool practical.

7.3.1 Merits

We presented an efficient encoding of Java in Alloy that overcomes shortcomings of the common idioms for expressing object-oriented models. In particular, the encod-

ing avoids the scope-induced explosion problem, where one or more types require a large number of atoms. We achieved this by introducing a variable renaming similar to single static assignment. Despite the fact that our encoding adds relations, it nevertheless has a tighter translation to propositional logic than the common idioms.

Our encoding uses edge variables – booleans that label each edge of the computation graph. These simplify the translation of Java to Alloy, but they are also used to report counterexamples back to the user. Their truth values in an instance indicate which edges of the computation graph were traversed.

The optimizations we presented help to improve the scalability of the analysis, and allow larger procedures to be checked with fewer annotations. They target a common case: field dereferences. They do not provide any improvement in scalability when a fragment of code does not have many field dereferences. The optimizations are based on a compact functional representation for fields, and aim at reducing the size of the CNF produced. They are synergistic in that implemented alone many of them do not result in an improvement. Although inspired by the verification of code, they are in fact more general. The compact representation is applicable to any Alloy model containing functions. The logical simplifications can be used independently in other contexts.

7.3.2 Deficiencies

The thesis has several shortcomings. First, the prototype tool supports a very limited subset of Java. For some omitted constructs, we described how the analysis might handle them. This is the case for subclassing and user-defined exceptions, as well as uni-dimensional arrays. The shortcoming here is that due to the lack of experimentation, the efficiency of the resulting analyses remains to be seen.

Other constructs are not treated at all. These include built-in exceptions other than null-pointer dereferences; multi-dimensional arrays; and multi-threading. There does not seem to be any inherent obstacle in handling concurrency: one might simply represent all possible interleavings in the computation graph. This would result in a larger encoding for the control flow. So the efficiency of the resulting analysis also remains to be seen.

A major omission from the subset of Java supported is recursion. We could handle singly-recursive procedures by unwinding them as we do to loops. Mutually recursive procedures may be handled in a similar manner.

We represent null as the empty set, which prevents us from analyzing code that has collections containing null. This would not have been the case if we had represented it using a special null object. However, this would require treating all objects as members of the type Object, dramatically increasing the scope (and thus damaging the performance of the analysis). The latest version of Alloy alleviates this problem by offering subtypes and union types, and allowing scopes to be set on individual subtypes. A variable would then be declared as being of its own type unioned with null, and collections would be able to contain null explicitly. Classes would be represented as subtypes of Object, and could be bounded independently.

A second major shortcoming is the lack of a specification language for our analysis

technique. Currently we use Alloy itself without any kind of veneer, which is not ideal. A practical specification language would benefit from a "modifies" clause, for example, allowing frame conditions to be expressed. There are also a number of minor but important details to overcome. For example, because Alloy has no overloading on function names, Jalloy cannot distinguish Java methods with the same name. With a more expressive specification language, a wider variety of checks might also be offered. For example, one may want to check that, if one procedure runs followed by another, some property holds.

Currently, there are very few specifications built-in for the Java API. More of these are needed to make the tool practical. For the specifications we provided we omitted dealing with methods that provide views, which abstract aspects of a data type, and lead to a kind of "abstract aliasing".

A final shortcoming of the thesis is that there are very few experiments. Although the experiments we performed do illustrate the feasibility of the analysis technique, as well the efficacy of the optimizations, they are not sufficient to evaluate ease-of-use. More experiments are needed to show that our technique is easy to use in practice. A better prototype tool would help in this direction.

7.3.3 Future Opportunities

The thesis offers some opportunities that were not realized due to lack of time. These include incorporating the optimizations in the Alloy Analyzer. Currently, the prototype tool translates Java directly to CNF and the result is conjoined with the translation of the specification to CNF produced by the Analyzer. So the translation to Java does not benefit from the Analyzer's simplifications and optimizations such as symmetry-breaking [36], and subformula sharing [37].

One missed opportunity is the treatment of null-pointer dereferences in the way we presented in an earlier paper [19]. This approach is more efficient than the one we presented here, but was not easy to handle by our direct translation of Java to Alloy. Once the optimizations are incorporated in the Analyzer, it could easily be put in place.

Recall that frame conditions are needed at join points in the computation graph to unify the names of variables. These formulas can potentially be large, and we alleviated this problem by ignoring variables representing subexpressions not needed beyond the join point. This might be taken further, by avoiding the propagation of the name of any non-live variable – that is, whose value is not needed beyond a certain point. No doubt other traditional compiler analyses could also be brought to bear on the problem of obtaining a compact representation.

Another opportunity is to use a technique called *type-splitting* in the analysis of code containing subclasses. This technique is not novel and has been used in other tools (e.g. [16]). It consists of splitting a type into more than one subtype, and assigning different subtypes to variables pointing to structures that do not share objects. For example, if we have two linked lists and we know that they do not share elements, we can assign a different subtype to each of the lists. This technique is helpful because it helps to reduce the size of the encoding in propositional logic: the

subtypes now have fields over smaller domains.

There are many opportunities for tool and user interface improvements. In our prototype, specifications are provided in a separate file; it would be more practical to allow in-code annotations. The output trace is currently displayed by highlighting the code in an emacs window using a script. A graphical user interface could show heap snapshots pictorially (using a facility similar to the Alloy Analyzer's visualizer), correlating them with program points.

Finally, there are complementary approaches that would improve scalability. The optimizations we presented are bottom-up: we optimized the Analyzer for the verification of code. Another approach is to treat the Analyzer as a black box and to perform abstractions of the code that allow larger fragments to be analyzed. Conventional techniques such as slicing might play a role. Procedural abstractions might be refined incrementally: replacing a procedure with a specification that allows all possible behaviors and successively refining it [38]. Such techniques could be fruitfully combined with ours.

7.4 Final Thoughts

Programmers often shy away from writing structural specifications, even though they are invaluable for understanding code and finding bugs. This state of affairs may be due to the lack of a tool for checking specifications. Theorem proving can be used to verify structural properties, but is hard to automate fully. Existing finite-state verification techniques do not handle structure readily. Other automated tools require many intermediate annotations beyond the specification to verify.

Our objective was to build a finite-state verification system for checking structural specifications. The technique we presented is fully automatic, modular, and does not require any user-provided abstractions. The only abstraction it uses is to consider bounded instances of code. It outputs sound counterexamples.

Modularity comes at a price. The user may write intermediate specifications for procedure calls, but we have aimed to minimize the need for this. For the Java API, our technique has some specifications built-in. To minimize intermediate annotations and allow larger fragments of code to be checked, we presented a series of synergistic optimizations that reduce the size of the CNF produced exponentially. Although the optimizations were inspired by the verification of Java code, they are in fact more general. They can be applied to any Alloy model containing functional relations. The logical simplifications can also be used independently.

Our case studies demonstrated the effectiveness of the optimizations in practice, as well as the usefulness of our technique in finding real bugs. The size of procedures does not grow very much over time, but processors get faster and faster. Our technique, therefore, should become more effective as time passes. Our hope is that it will also be successful in promoting the use of specifications as an invaluable tool for finding bugs in software systems.

Bibliography

[1] A. Andoni, D. Daniliuc, S. Khurshid and D. Marinov. Evaluating the "Small Scope Hypothesis". Unpublished manuscript. September 2002.

[2] T. Ball, S. K. Rajamani. "The SLAM Project: Debugging System Software via Static Analysis", *Proc. Principles of Programming Languages (POPL'02)*, January 2002.

[3] W. R. Bush, J. D. Pincus, and D. Sielaff. "A Static Analyzer for Finding Dynamic Programming Errors", Software Practice and Experience 2000 (30):775–802.

[4] D. R. Chase, M. Wegman and F. Zadeck. "Analysis of Pointers and Structures", *Proc. Programming Language Design and Implementation*, 1990.

[5] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, H. Zheng. "Bandera: Extracting Finite-State Models from Java Source Code", *Proc. International Conference on Software Engineering*, June 2000.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest. "Introduction to Algorithms", MIT Press, 1990.

[7] D. Detlefs, K. R. Leino, G. Nelson, and J. Saxe. "Extended Static Checking". Technical Report 159, Compaq Systems Research Center, 1998.

[8] E. Dikjstra. "A Discipline of Programming". Prentice-Hall, 1976.

[9] C. Flanagan, R. Joshi, X. Ou, and J. Saxe. "Theorem Proving using Lazy Proof Explication", *Proc. Computer-Aided Verification 2003 (CAV'03)*, Boulder, Colorado, July 2003.

[10] E. Goldberg and Y. Novikov. "BerkMin: A fast and robust SAT-solver", *In Design, Automation, and Test in Europe*, March 2002.

[11] S. Graf and H. Saidi. "Construction of abstract state-graphs with PVS", *Proc. Computer-aided Verification (CAV'97)*, LNCS 1254, pages 72–83, 1997.

[12] G.J. Holzmann. "The Model Checker Spin", *IEEE Trans. on Software Engineering*, Vol. 23, 5, May 1997.

[13] G. J. Holzmann and M. H. Smith. "Automating Software Feature Verification", *Bell Labs Technical Journal*, Vol. 5, 2, April-June 2000.

[14] D. Jackson. "Automating First-Order Relational Logic", *Proc. Foundations of Software Engineering*, San Diego, November 2000.

[15] D. Jackson and A. Fekete. "Lightweight Analysis of Object Interactions", *Proc. Fourth International Symposium of Theoretical Aspects of Computer Software*, Sendai, Japan, October 2001.

[16] D. Jackson, S. Jha, and C. Damon: "Faster Checking of Software Specifications by Eliminating Isomorphs", *Proc. Principles of Programming Languages (POPL'96)*, 1996.

[17] D. Jackson, I. Shlyakhter and M. Sridharan. "A Micromodularity Mechanism", *Proc. Foundations of Software Engineering*, 2001.

[18] D. Jackson and K. Sullivan. "COM Revisited: Tool Assisted Modelling and Analysis of Software Structures", *Proc. Foundations of Software Engineering*, San Diego, California, November 2000

[19] D. Jackson and M. Vaziri. "Finding Bugs with a Constraint Solver", *Proc. International Symposium on Software Testing and Analysis (ISSTA'00)*, Portland, Oregon, August 2000.

[20] S. Khurshid and D. Jackson. "Exploring the Design of an Intentional Naming Scheme with an Automatic Constraint Analyzer", *Proc. 15th IEEE International Conference on Automated Software Engineering*, Grenoble, France, September 2000.

[21] S. Khurshid, C. Pasareanu, and W. Visser. "Generalized Symbolic Execution for Model Checking and Testing", *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, Warsaw, Poland, April 2003.

[22] T. Lev-Ami and M. Sagiv. "TVLA: A System for Implementing Static Analyses", *Proc. Static Analysis Symposium*, 2000.

[23] B. Liskov with J. Guttag. "Program Development in Java. Abstraction, Specification, and Object-Oriented Design", Addison-Wesley, 2001.

[24] R. Manevich, G. Ramalingam, J. Field, D. Goyal, M. Sagiv. "Compactly Representing First-Order Structures for Static Analysis", *Proc. Static Analysis Symposium (SAS'02)*, 2002.

[25] D. Marinov and S. Khurshid. "TestEra: A Novel Framework for Automated Testing of Java Programs", *Proc. Automated Software Engineering (ASE'01)*, Nov 2001.

[26] K. L. McMillan. "Symbolic model checking - an approach to the state explosion problem", PhD thesis, SCS, Carnegie Mellon University, 1992.

[27] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, S. Malik. "Chaff: Engineering and Efficient SAT Solver", *Proc. Design and Automation Conference*, Las Vegas, June 2001.

[28] G. Nelson and D. Oppen. "Simplification by cooperating decision procedures", *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.

[29] T. Nolte. "Exploring Filesystem Synchronization with Lightweight Modeling and Analysis", Masters thesis, MIT Laboratory for Computer Science, Cambridge, MA, August 2002.

[30] M. Vaziri and T. Nolte. "Augmenting a Finite-State Verification System with a Decision Procedure", submitted for publication, October 2003.

[31] The Omega Library Web Page: `http://www.cs.umd.edu/projects/omega/`

[32] D. A. Plaisted and S. Greenbaum. "A Structure-Preserving Clause Form Translation", *Journal of Symbolic Computation*, 2:293–304, 1986.

[33] 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.

[34] SAT Competitions. `http://www.satlive.org/SATCompetition`.

[35] R. Seater. "Alloy 2.0 Tutorial", `http://alloy.mit.edu`.

[36] I. Shlyakhter. "Generating Effective Symmetry-Breaking Predicates for Search Problems", *SAT'01 Workshop*, Electronic Notes in Discrete Mathematics, Vol. 9, June 2001.

[37] I. Shlyakhter, M. Sridharan, R. Seater, D. Jackson. "Exploiting Subformula Sharing in Automatic Analysis of Quantified Formulas", *Proc. Conference on Theory and Applications of Satisfiability Testing (SAT'03)*, May 2003.

[38] M. Taghdiri. Personal communication.

[39] M. Taghdiri. "Lightweight Modelling and Automatic Analysis of Multicast Key Management Schemes", Masters Thesis, MIT Laboratory for Computer Science, Cambridge, MA, December 2002.

MIT Laboratory for Computer Science, manuscript, March 2003.

[40] M. Vaziri and D. Jackson. "Checking Properties of Heap-Manipulating Procedures with a Constraint Solver", *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, Warsaw, Poland, April 2003.

[41] W. Visser, K. Havelund, G. Brat and S. Park. "Model Checking Programs", *Proc. International Conference on Automated Software Engineering*, September 2000.