November 12, 2018

INTEGRATING FORMAL METHODS TOOLS

A CASE STUDY

Author
Matthijs Hofstra

Faculty
Electrical Engineering, Mathematics & Computer Science

Chair
Formal Methods & Tools

Committee
prof. dr. Marieke Huisman
dr. Ansgar Fehnker

UNIVERSITY OF TWENTE.

Preface & Acknowledgements

This thesis is written as part of my final project of my Computer Science master at the University of Twente.

I would like to thank my committee: Marieke Huisman for her support, feedback, and considerable patience, and Ansgar Fehnker for joining the committee after changes in the scope and focus of my research called for a change of committee.

Abstract

Many formal methods tools exist that can assist in the creation of correct software programs, and more are created each year. Yet many of these tools focus only on one specific technique, and are unable to deal with problems that are unsuited for that technique. We believe that combining tools that implement different techniques can yield great results. Yet it seems that little research is being done in this area. We suspect that one of the reasons is that formal methods tools tend to be hard to integrate with existing software. If integrating different formal methods tools were easier, then the barrier to combine different tools would be much lower. As a secondary effect, when tools are easier to integrate it will be easier for them to be adopted outside the academic world.

In this thesis we will explore the difficulty of integrating different formal methods tools by performing a case study in which we will create a framework for generating loop invariants for Java programs. The tools for generating loop invariants will serve as a hopefully representative sample of formal methods tools in general. The framework will integrate several existing tools that either perform automated loop invariant generation or contribute to automated loop invariant generation. We will report on the problems encountered during the integration of the different tools and investigate how these problems could have been avoided. Based on this analysis we will create guidelines that can be used by formal methods researchers to make the results of their work more accessible to others. The ultimate aim of these guidelines is twofold: to facilitate research by making it easier for formal methods researchers to build on the work of others, and to allow research results to make the jump from the academic world into the software development industry.

Keywords:
Loop Invariant Generation, JML, Integrating, Formal Methods Tools

Table of Contents

1 Introduction **11**

1.1 Background 11

1.2 Goals 12

1.3 Thesis organization 13

2 The Java Modeling Language **15**

2.1 JML annotations on classes and fields 15

2.2 JML annotations on methods 16

2.3 JML annotations inside methods 17

2.4 Checking JML 18

2.5 OpenJML 19

2.6 Summary 19

3 Loop Invariant Generation **21**

3.1 Polynomial Invariant Generation by Algebraic Techniques 21

3.2 Inferring Invariants in Separation Logic for Imperative List-processing Programs . 22

3.3 Daikon 24

3.4 DynaMate 25

3.4.1 EvoSuite 26

3.4.2 GIN-DYN 26

3.5 Summary 27

4 Setting up the Case Study **29**

4.1 Goal 29

4.2 Requirements . . . 29
4.2.1 The framework should integrate several different tools . . . 30
4.2.2 The framework should reuse existing functionality whenever possible . . . 30
4.2.3 The framework should be extensible . . . 30
4.3 Conclusion . . . 30

5 Architecture 31

5.1 Framework design . . . 32
5.1.1 Test-case generation . . . 32
5.1.2 Dynamic invariant detection . . . 32
5.1.3 Static analyzers . . . 33
5.1.4 Invariant checkers . . . 33
5.2 Components . . . 34
5.2.1 ExecutionProvider . . . 34
5.2.2 DynamicInvariantGenerator . . . 34
5.2.3 StaticInvariantGenerator . . . 34
5.2.4 InvariantReject . . . 35
5.3 Operation of the framework . . . 35

6 Implementation 37

6.1 Building the Context . . . 37
6.2 EvoSuite . . . 38
6.3 Basic static invariant generator . . . 38
6.4 Daikon . . . 38
6.4.1 Gathering loop information . . . 38
6.4.2 Interpreting the results . . . 39
6.5 OpenJML . . . 40

7 Findings 41

7.1 Documentation . . . 41

7.2 Tools may not clean up properly after running . . . 42
7.3 Modifying the AST in OpenJML . . . 42
7.4 Interactions between EvoSuite and Daikon . . . 43

8 Advice **45**

8.1 Build tools as a library . . . 45
8.2 Have technical documentation on using the API . . . 46
8.3 Avoid global state . . . 46
8.4 Use common libraries . . . 46
8.5 Future work . . . 47
8.5.1 Investigate common needs in research tool development . . . 47
8.5.2 Further research into combining the results of multiple loop invariant generation tools . . . 47
8.5.3 Investigate combining tools in other areas in formal methods . . . 47
8.6 Conclusion . . . 48

Chapter 1

Introduction

In this chapter we will give a brief background of the circumstances motivating our research. This will be followed by the goals of our research, and an overview of the organization of this thesis.

1.1 Background

Each year a lot of research into formal methods is performed in the field of computer science. New techniques are investigated, existing techniques are refined, and tools are created to test and demonstrate the usefulness of these techniques. While much of this work is interesting in a theoretical sense, the ultimate goal of formal methods is to enable the creation of more reliable and robust systems.

While all this research is taking place, software is growing in importance. Computers are no longer just large devices standing on a desk; they surround us at all times. We carry them in our pockets as mobile phones, and some people carry them on their wrists as smart watches. A lot of communication that used to happen in person or by writing a letter now happens electronically over the internet.

These developments are not without risks. A system that can be used over the internet may also be abused over the internet. The more technology becomes a part of our lives, the more we are affected when that technology fails. While in some situations the failure of technology is merely inconvenient, in other contexts the consequences can be much more severe, ranging from loss of privacy and financial damage to risk of bodily harm or even death.

Some industries have adopted formal methods as part of their process, but many have not. The damage caused by software bugs is estimated to cost billions of dollars annually. In light of such numbers one would expect there to be more incentive for the industry as a whole to adopt formal methods, but to date this has not happened.

There are many possible reasons for the failure of the industry to adopt formal methods. Some may have doubts about the applicability or (cost-) effectiveness of formal methods. Perhaps there is a lack of knowledge about what formal methods have to offer. Others reasons may be more technical in nature, such as formal methods tools being difficult to use, or hard to

include in existing software development workflows.

We believe that one of the problems facing the field of formal methods is the difficulty of integrating formal methods tools in other software. This problem negatively affects both the ability of researchers to build on the work of others, as well as the ability for the industry to incorporate formal methods tools in software development workflows, for example through integration in integrated development environments. In this thesis we will focus on the integration of formal methods tools with other formal methods tools, but we hope that improvements in this area will also facilitate increased industry adoption.

When a new formal methods tool is created and a paper is written, the ease with which this new tool can be integrated within existing software is unlikely to be one of the most important qualities that this new work will be judged on, both by the authors and peer reviewers. This is unsurprising because "can be easily integrated with other software" is not a quality that has academic value. The produced software is often a proof-of-concept; it will contain the required functionality to show the effectiveness and viability of the ideas of the author(s), but lack the polish that would make it usable in different situations.

1.2 Goals

The ultimate goal of our research is the increased use of formal methods outside of the academic world, because we believe that increased real-world use of of formal methods will be beneficial for both the industry and the field of formal methods itself. We hypothesize that one of the reasons that formal methods tools are slow to gain adoption in the industry is because there are (possibly unnecessary) technical hurdles to integrate these tools within other software. We will investigate the existence of these technical hurdles and what can be done to overcome them. We hope solving these problems will ultimately lead to increased use of formal methods tools in the industry.

A large step towards this goal would be for the field of formal methods itself to gain an increased interest for combining different tools. The work by Galeotti et al. on DynaMate [10] has shown that combining different tools can be very effective, because tools can have different strengths, and combining different tools can solve problems that none of the individual tools could have solved. But we believe that the difficulty of integrating different tools is standing in the way of further research into combining tools.

To be able to identify the technical difficulties that are encountered when integrating formal methods tools in other software, we would like our case study to be a "realistic" instance of such an integration. We believe loop invariant generation to be a great subject to use for our case study, because many different techniques exist.

Therefore we have chosen the implementation of a framework for the integration of different loop invariant generation tools as our case study. We believe the creation of this framework gives a good impression of the technical difficulties that can be encountered when integrating formal methods tools, because various different tools will be involved. Because the results of the different tools must be combined and sometimes passed to other tools, this forces a real integration, where the framework must understand the various inputs and outputs of the tools

that are used.

1.3 Thesis organization

In chapter 2 we will describe the Java Modeling Language. This language can be used to describe properties of a Java program. Program properties most relevant to our study will be method preconditions, method postconditions, and loop invariants. Our description of the Java Modeling Language is incomplete, but covers a subset that is sufficient to understand our research.

Chapter 3 describes several existing tools and techniques for automatic loop invariant generation, to give an impression of the state of the art. While too much work has been done in the field of loop invariant generation to summarize it all in a single chapter, all the loop-invariant-generating tools that we use in the case study are included in this chapter.

In chapter 4 we describe the requirements of our framework, and motivate these requirements based on the goal of our research.

Chapter 5 gives a description of the architecture of our framework. We define several categories of tools related to loop invariant generation, and describe how tools in these categories can work together in the framework.

Chapter 6 gives an overview of the actual implementation of the framework. It describes how the various tools are used, including how they are given their inputs, and how their outputs are interpreted.

In chapter 7 we describe the problems we encountered while implementing the framework and combining the different tools.

In chapter 8 we give recommendations for tool creators that would help make their tools more suitable for integration. This advice will be based on our experiences while creating the framework, and on the problems we encountered in chapter 7. After the advice we will give suggestions for future work, and present our conclusions.

Chapter 2

The Java Modeling Language

The Java Modeling Language, hereafter referred to as 'JML', is a language that can be used to describe the behavior of Java programs. This behavior is described in a 'design by contract' style, where a programmer can specify invariants of a class and the preconditions and post-conditions of methods and constructors. Methods and constructors are treated very similarly in JML, with the necessary exception that class invariants and constraints need not hold on entry of a constructor, and are implicitly postconditions of constructors. The distinction between constructors and methods will not be very relevant to our research and from this point on we will refer to both as simply *methods*, unless specified otherwise. Below we will discuss several keywords from JML that we expect will be most relevant to our research, and we will give some examples that demonstrate their use. For a complete description of JML we recommend the JML Reference Manual [13], which describes the language in detail.

2.1 JML annotations on classes and fields

Specifying a class invariant happens with the `invariant` keyword. It is not required for an invariant to hold at all times, as long as it always appears to hold from the perspective of an outside observer. JML accomplishes this by distinguishing between visible states and invisible states of an object, where visible states include states such as 'at the end of a constructor' and 'at the start and end of a method', and invisible states include states such as 'at the start of a constructor' and 'while inside a method'. An exception exists for private methods and private constructors that are annotated with the `helper` keyword. Helpers are special because invariants do not have to hold on entry, nor do they need to be established on exit. Because these helpers are private and can therefore be called only by other methods and constructors of the class (that *do* have to ensure invariants) this does not result in observable invariant violations.

In addition to invariants, JML has constraints, which are specified with the `constraint` keyword. Constraints relate states in the lifetime of an instance. For example, a constraint could specify that the value of a member variable can only increase. For any two visible states in an objects lifetime, its constraints constrain how the state of the instance can change between the older state and the newer. Below is an example that demonstrates an invariant and a constraint. In the example, an invariant is used to state that the member field `n` must

always be larger or equal to zero. A constraint is used to state that the value of `n` must always be equal or greater than in any states that came before, in other words, it can never decrease. In order to refer to the private `n` in the specification, we use the `spec_public` keyword, which allows us to refer to private and protected fields in the specification as if they were public. In addition, we use `\old(E)` keyword, which in this context allows us to refer to the value of an expression `E` in a previous state. When using `\old(E)` in a method postcondition or in a method body, it evaluates to the value that `E` had on entering that method.

```
1 class Example {
2     private /*@ spec_public @*/ int n;
3     //@ public invariant n >= 0;
4     //@ public constraint n >= \old(n);
5 }
```

2.2 JML annotations on methods

As mentioned earlier, JML also allows a programmer to specify the behavior of methods, following the design by contract paradigm. There are many things that we can say about a method using JML, the most basic of which are its pre- and postconditions. The `requires` keyword is used to specify a precondition, and the `ensures` keyword is used to specify a postcondition. A basic example that demonstrates these two keywords can be found below.

```
1     //@ requires n < Integer.MAX_VALUE;
2     //@ ensures \result == n + 1;
3     public int addOne(final int n) {
4         return n + 1;
5     }
```

JML annotations can also specify other behavioral aspects, such as under which conditions exceptions are thrown (using the `signals` keyword), which fields may be modified by a method (using the `assignable` keyword), and if a method diverges or not (using the `diverges` keyword). The example below demonstrates this by showing the specification of a method that takes an integer `n`. When `n` is negative, an exception is thrown. Otherwise, when `n != 12` we return the number 8, and when `n == 12`, we loop forever.

```
1      /*@ normal_behavior
2        @   requires n >= 0;
3        @   ensures \result == 8;
4        @   assignable \nothing;
5        @   diverges n == 12;
6        @ also
7        @   exceptional_behavior
8        @   requires n < 0;
9        @   assignable \nothing;
10       @   signals (Exception) true;
11       @*/
12     public int foo(final int n) throws Exception {
13         if (n < 0)
14             throw new Exception();
15         if (n != 12)
16             return 8;
17         while (true) {}
```

```
18      }
```

In the above example we used heavyweight specification, and in the example before that we used lightweight specification. The heavyweight specification is more expressive and we suspect that this is the type of specification that we will use in the actual project, but the lightweight specification is more succinct and thus useful for examples. The syntax and semantics of loop invariants are the same for both types of specification.

2.3 JML annotations inside methods

In addition to making it possible to describe the behavior of a class and its methods from the perspective of a user of that class, JML can also be used to describe behavior inside a method. This can be done using assertions and loop invariants. An assertion states that a certain predicate should always hold whenever program execution reaches the point where the assertion is defined. In JML the `assert` keyword is used to specify an assertion. Note that the Java programming language also has an `assert` keyword, however because it cannot guarantee that evaluating its predicate is free from side-effects and because it is less expressive than `assert` in JML, we will not use it. A loop invariant can be thought of as an assertion that has to hold just before entering a loop, and at the end of the body of a loop. This is useful because a correct loop invariant allows us to say things about the properties of a loop without knowing the number of iterations in advance. While assertions and loop invariants are usually implementation details that are unimportant to users, they can be valuable to programmers. Even without tool support, assertions and loop invariants can be used to document the assumptions that were made by the author. With tool support, there are two more ways that we can use assertions and loop invariants: we can verify that they hold at run-time using run-time assertion checking, or use them in combination with static analysis to prove a method correct. For static analysis loop invariants are essential to proving that a method obeys its specification. Both of these applications will be discussed later in this chapter.

In Figure 2.1 we demonstrate a use of the `loop_invariant` and `assert` keywords. The `allTrue` method checks if an array of booleans only contains `true`. To do this, we iterate over the array, accumulating the result in a variable named `result`. We add an invariant to the loop that describes this behavior: we state that the value of `result` is equal to the property of all positions of array `arr` with an index smaller than `i` being `true`. This property holds trivially on loop entry; there are no indices smaller than `i` because `i = 0`, so the property is vacuously true. In the loop body we check the current position in the array, and increase `i` by one, thus once again establishing the loop invariant. Because the loop invariant holds on loop entry and at the end of every execution of the loop body, when we exit the loop we know that the invariant holds, and that the loop condition is false. In the example, this knowledge is made explicit using the `assert` keyword, and the method postcondition is easily established.

```
/*@ requires arr != null;
  @ ensures \result ==
  @    (\forall int i; 0 <= i && i < arr.length ; arr[i]);
  @*/
public static boolean allTrue(final boolean[] arr) {
    boolean result = true;
    int i = 0;
    /*@ assert arr.length >= 0;
      @ loop_invariant result ==
      @    (\forall int j; 0 <= j && j < i ; arr[j]);
      @ loop_invariant i >= 0 && i <= arr.length;
      @*/
    while (i < arr.length) {
        result = result && arr[i];
        i++;
    }
    /*@ assert result ==
      @    (\forall int j; 0 <= j && j < i ; arr[j]);
      @ assert i == arr.length;
      @*/
    return result;
}
```

Figure 2.1: An annotated method demonstrating preconditions, postconditions, loop invariants, and assertions.

2.4 Checking JML

Various tools exist that can check if a piece of Java code actually matches its JML specification. They can be divided in two categories: tools that check specifications at run-time (*run-time assertion checking*), such as *jmlrac*, and tools that prove that the specification is correct by examining the source code (*static checking*), such as *ESC/Java2* and *OpenJML*.

The tool *jmlrac* comes with *jmlc*, a program which can compile Java source files into Java class files like a normal Java compiler, but it also recognizes JML and will generate extra code that will check that the specification is obeyed at run-time. This checking includes preconditions (`requires`), postconditions (`ensures`), assertions, invariants, loop invariants, etcetera. This way, when you execute the resulting Java code (either by running it normally or when running tests) an exception will be thrown if the execution does not match the specification.

On the other hand, tools such as `ESC/Java2` and `OpenJML` take the JML specification and the Java code and attempt to prove (using an automated theorem prover) that the code obeys the specification. This does not require compiling and executing the Java code, although large parts of the process of translating the annotated Java code to input for a theorem prover is very similar to the work performed by an ordinary compiler.

Both approaches have their advantages and disadvantages. For example, with run-time checking you can omit loop invariants completely and it will still be possible to check method preconditions and postconditions. However, run-time checking will only tell you if the specification is correct for the inputs that you have tried. In contrast, static checking will let you prove that methods obey their specification, but the tools that can do this usually require more guid-

ance, especially in the form of loop invariants for code with loops. One important property of loop invariants is that they cannot be automatically derived in the general case. Static checking tools also tend to have problems with reasoning about floating point arithmetic, and they may ignore the possibility of integer overflow.

2.5 OpenJML

OpenJML is a tool set that can help verify JML specifications. It supports both static checking and run-time checking for Java up to version 1.7, and was created to replace ESC/Java2, a similar tool which supports Java 1.4. OpenJML is built on OpenJDK, an open-source Java Development Environment, by modifying and extending the OpenJDK Java compiler. For example, parsing of JML specifications is added to the parsing phase, and before the code generation there is now the option of first modifying the AST to insert run-time checks, or to perform static checking and skip code generation entirely.

One of the key goals of OpenJML is to provide challenge problems to SMT solvers [3], which it uses for its static checking. OpenJML has been tested with Z3 [4], CVC4 [1], and Yices [5], and it uses these solvers by translating Java methods and their specifications to a verification condition and presenting this to a solver. The verification conditions are expressed in the SMT-LIBv2 format, which is a standard input format that all the aforementioned solvers support. The solver can then reason about the problem and give the result to OpenJML, which then informs the user of the result. Because OpenJML keeps a mapping between the original Java code and the verification conditions it generates, it can translate the results of the SMT solver into a message that a user can understand. If the SMT solver finds a counter-example that demonstrates a violation of the specification then OpenJML can print information about the values that variables took to reach the state where the specification was violated.

When OpenJML performs its static checking it will evaluate each method in isolation, replacing calls to other methods by an assertion of its preconditions and and assumption of its postconditions. This is convenient because a method that fails to verify (e.g. because it is missing a loop invariant) does not preclude other methods from being verified, as long as it has correct pre- and postconditions. This means that when adding JML specifications to an existing codebase, it is not required to annotate the entire codebase all at once to get useful results.

2.6 Summary

JML is a language that can be used to annotate Java programs. These annotations can formally specify the behaviour of the annotated code. JML is able to describe many program properties. The properties that are most important to this thesis are method preconditions, method postconditions, and loop invariants, all of which are demonstrated in Figure 2.1. Tools such as OpenJML are able to use static analysis to prove that a Java program obeys its JML specification, provided that the specification is correct. Loops are hard for tools to reason about and are generally required to be annotated with loop invariants for a correctness proof to succeed.

Chapter 3

Loop Invariant Generation

As described in the previous chapter, tools performing static analysis to prove that a program obeys its specification require loop invariants to reason about loops. These loop invariants cannot be automatically derived in the general case. This limitation has not stopped formal methods researchers from creating tools and techniques that are able to derive loop invariants in specific cases. In this chapter we will describe several of these tools and techniques. Several of the tools described in this chapter are used in the loop invariant generation framework we have created.

3.1 Polynomial Invariant Generation by Algebraic Techniques

Kovács and Jebelean identify a subset of imperative loops called *P-solvable* loops [11] [12]. An algorithm is presented that can generate a finite set of polynomial relations among the loop variables, provided that the loop body contains only assignments and no conditionals. From these polynomial relations any polynomial invariant can be derived. It is not clear if this method is complete for P-solvable loops with conditionals, but without conditionals it is complete.

The algorithm works by expressing the value of each variable in the loop body as a recurrence equation. So the value of a variable in iteration n can be expressed as a formula $f(n)$ that depends on the values of variables in iteration $n-1$. Naturally $0 \leq n$, and $f(0)$ is defined to be the value of a variable after the first iteration. If a loop body contains conditionals, these are translated to more loops. For example, the program in figure 3.1 can be translated to the program in figure 3.2 (in these examples, e_i and c_j are expressions and statements respectively). When all the conditionals are removed, the loop bodies are analyzed from the most inner loops to the most outer loops by solving the recurrence relations. This yields a loop invariant that holds for the outer loops *and* any inner loops.

```
While (e1) {
    c1;
    if (e2) { c3 } [else { c4 }];
    c5;
}
```

Figure 3.1: A loop containing a conditional

```
While (e_1) {
    While (e_1 ∧ e_2) { c_1; c_3; c_5 }
    While (e_1 ∧ ¬e_2) { c_1; [c_4;]c_5 }
}
```

Figure 3.2: The loop with the conditional replaced by inner loops.

Kovács and Jebelean describe how the algorithm is able to find interesting loop invariants for several nontrivial programs containing loops, such as a program for calculating square roots by E.W. Dijkstra, a program for calculating square roots by K. Zuse, and a program for calculating the product of two numbers.

The algorithm appears to work very well for loops that manipulate numbers, but is unable to work with non-numeric data structures. A tool named Valigator has been created, which implements their algorithm.

3.2 Inferring Invariants in Separation Logic for Imperative List-processing Programs

Magill et al. describe a way to reason about the shape of data and derive loop invariants using separation logic [14]. Their algorithm works for (potentially cyclic) linked lists and was able to extract loop invariants automatically for several pointer programs that perform destructive heap operations. A list is modeled using a recursive definition that can easily be folded and unfolded. Their technique will symbolically execute a loop body multiple times, and for each iteration attempt to fold into lists all memory cells that are not directly pointed to by a program variable. When the algorithm finds a fixed point it can generate an invariant. The technique in the paper has only the boolean, integer, and list-of-integers types, but the authors believe that their technique can be extended to allow types such as list-of-booleans and lists-of-lists.

The authors use the two predicates below to describe a list. The first describes a list that may or may not be empty, while the second describes a non-empty list. The notation $a \longmapsto (b, c)$ means that a points to a tuple of values b and c. This structure is used to represent a list by having the first value of the tuple represent a value, and the second value represent a pointer to the remainder of the list.

$$\begin{aligned} \mathsf{ls}(p_1, p_2) &\equiv (\exists x, k \,.\, p_1 \longmapsto (x, k) * \mathsf{ls}(k, p_2)) \lor \\ &\qquad (p_1 = p_2) \land \textbf{emp} \end{aligned}$$

$$\mathsf{ls}^1(p_1, p_2) \equiv \mathsf{ls}(p_1, p_2) \land p_1 \neq p_2$$

During symbolic execution, memory is represented as a triple $(H; S; P)$. At any point there can be multiple possible states that the memory may be in, so we keep track of a set of possible memories $\{(H; S; P)\}$. Entities in the heap are expressed in separation logic and listed in H, assignments on the stack are listed in S, and P contains predicates in classical logic, augmented with arithmetic. The symbolic evaluator starts with the annotation

$H \wedge P$, which describes the initial state of the heap and properties of the stack variables. The authors illustrate the workings of their method using an example program, we will use the same program, but translated to Java. Note that the `Node.Next` property should be considered to be equal to the k from the ls definition above (because we never inspect the contents of the list we can ignore x in this case). This program can be seen in figure 3.3. For this program, the starting annotation would be $\mathsf{ls}(oldList, null)$, yielding an initial state of $\{\exists v_1, v_2, v_3.(\mathsf{ls}(v_1, null); oldList = v_1, newList = v_2, tmp = v_3;)\}$ (P is empty). Note that from this point on we will omit existential quantifiers. The authors have defined symbolic evaluation rules that manipulate the state, we will not repeat them here, but they are fairly straightforward. The rules take a single memory or a set of possible memories, and produce a memory or a set of memories. For example, after executing the first two statements of the example program, the state would be $\{(\mathsf{ls}(v_1, null); oldList = v_1, newList = null, tmp = v_1;)\}$. If required $\mathsf{ls}(a, b)$ can be unfolded to its right-hand side, but this is only done if progress is otherwise impossible because unfolding can be done an arbitrary number of times, which would blow up the state.

```
1  // Assume that Node is some class that has a 'Next' reference
2  // to another Node (or null), allowing construction of a singly
3  // linked list.
4  public static Node LinkedListReverse(Node oldList) {
5      Node newList = null;
6      Node tmp = oldList;
7
8      while (tmp != null) {
9          oldList = oldList.Next;
10         tmp.next = newList;
11         newList = tmp;
12         tmp = oldList;
13     }
14     return newList;
15 }
```

Figure 3.3: Translated example program from the paper

The algorithm handles loops as follows: when it enters a loop it adds the loop condition to the predicates P, then it symbolically executes the loop body. When it reaches the end of the loop body it attempts to fold the instances below:

$$v_a \longmapsto (x, v_b) * \mathsf{ls}(v_b, v_c)$$

and

$$\mathsf{ls}(v_a, v_b) * \mathsf{ls}(v_b, v_c)$$

into instances of:

$$\mathsf{ls}(v_a, v_c)$$

and

$$\mathsf{ls}(v_a, v_c)$$

Because folding loses some information it should not be applied in all cases. The authors have a heuristic where they do not fold away any value directly assigned to a program variable, so if S contains some assignment $x = v$ then v will not be folded away. This heuristic is not perfect and there exist programs that cannot be proved because too many folds are applied. A simple

example case mentioned by the authors is a program where the loop invariant relies on the property that some pointer is two steps ahead of another pointer: instances of $(v_1 \longmapsto (_, v_2) * v_2 \longmapsto (_, v_3) * \mathsf{ls}(v_3, v_4); x = v_1, y = v_3;)$ will be folded away to $(\mathsf{ls}(v_1, v_3) * \mathsf{ls}(v_3, v_4); x = v_1, y = v_3;)$ because v_2 was not assigned to any program variable. The program keeps executing the loop body until it reaches a fixed point, where the memory of the previous iteration implies the memory of the current iteration. If a fixed point is reached, the loop invariant is equal to the disjunction of the memories of each iteration; that is, if we number the memories for each iteration, where (0) is the memory before executing the loop body for the first time, (1) is the memory after executing the loop body one time, and (n) is the memory after executing the loop body n times, then if we reach the fixed point at iteration n, the loop invariant is equal to $(0) \vee (1) \vee (2) \vee \ldots \vee (n)$.

The example program reaches a fixed point after the second iteration. The resulting loop invariant is shown below. The first line is equal to the initial state, the third line is equal to the state after a single iteration, and the third line is equal to the state of all subsequent iterations.

$$\exists v_1.(\mathsf{ls}(v_1, \mathbf{null}); tmp = v_1, newList = \mathbf{null}, oldList = v_1)$$
$$\vee$$
$$\exists v_1, v_2, v_3.(v_1 \longmapsto (v_2, \mathbf{null}), \mathsf{ls}(v_3, \mathbf{null}); tmp = v_3, newList = v_1, oldList = v_3, v1 \neq \mathbf{null})$$
$$\vee$$
$$\exists v_1, v_3, v_5.(\mathsf{ls}^1(v_3, \mathbf{null}) * \mathsf{ls}(v_5, \mathbf{null}); tmp = v_5, newList = v_3, oldList = v_5; v_3 \neq \mathbf{null} \wedge v_1 \neq \mathbf{null})$$

There are cases where a fixed point cannot be found. For example, if all the values in a list are accumulated in some variable, we get a predicate of $sum = v_1 + v_2 + v_3 + \ldots + v_n$, potentially going on forever. Because the algorithm is primarily focused on shape information it deals with this problem by simply forgetting any information about integers, in the aforementioned case it would simply assign a fresh value to sum. The authors then propose to extend their method by adding predicate abstraction, which allows some information to be preserved while ensuring that the algorithm can still converge on a fixed point. While they have not implemented this extension, they claim that it proved useful when working out some examples by hand.

The authors describe in their results how their implementation of the algorithm allowed them to generate loop invariants for several programs performing list operations, such as computing the length of a list, summing the elements, deleting a list, reversing a list, and partitioning a list. Because the algorithm forgets integer arithmetic we assume that while in each of these cases some invariants were found, it cannot be the case that *all* invariants were found (for example, for calculating the length of a list or the sum of its elements), exactly which invariants were found is not clear from the paper.

3.3 Daikon

Ernst describes the tool Daikon, which performs dynamic program invariant detection. The tool is able to detect many invariants that hold during one or more executions of a piece of code, by monitoring the values taken by variables at specific points during execution [6] [7] [10]. This monitoring is usually performed while executing a test suite. By default Daikon does not attempt to find loop invariants. However, the tool will monitor which parameters are used

for method calls, so by inserting calls to dummy methods taking all interesting variables as their parameters at the entry and exit of a loop, the tool can be made to detect loop invariants.

The tool is divided in two parts: an instrumentor and the inference engine. The instrumentor performs monitoring either by inserting monitoring code into the source code of the program that is to be examined, or by operating directly on an executable. The inference engine (also named Daikon) will infer the invariants from the data produced by the instrumentor. Many languages, including Java, C, and C++, are supported, and adding support for a new language requires only the creation of an instrumentor for that language.

The inference engine is able to detect invariants such as a variable never being zero or $null$, a variable being constant, a variable being restricted to a certain range, a variable always being greater or smaller than another variable, and many more. Daikon will filter out redundant invariants that are consequences of stronger invariants, for example, if it finds that $x = 5$ and $10 <= y <= 30$, then it will not report invariants such as $x \neq 0$ and $x < y$, which are logical consequences. In addition to supporting various invariants over numeric variables, Daikon can also discover properties of sequences, such as one sequence being a subset or a reversal of another sequence, and a variable always having a value contained in a certain sequence.

In practice, it turns out that many of the invariants that are detected by Daikon turn out to be true invariants, holding not just in the limited number of executions that Daikon was able to examine, but in all possible executions. The quality of the test suite influences the quality of the detected invariants: a large test suite that tests methods with many different inputs that cover most of the corner cases will generate fewer spurious invariants than a test suite that often reuses the same input values and does not test corner cases.

While Daikon alone cannot ensure that the invariants it has found are generalizable, an automated prover can be used to verify that they apply to all possible executions. For instance, when using Daikon on a Java program the invariants can be added to a class or method as JML, and then a tool such as OpenJML can attempt to verify them.

3.4 DynaMate

Galeotti et al. describe how they created a framework named DynaMate for automated verification by combining several tools [10]. They test their framework on 28 methods containing loops from various `java.util` classes. Their framework manages to create complete correctness proofs for 25 of these methods. Below we will give an outline of the algorithm behind DynaMate, and then we will discuss the tools it uses.

The input of a DynaMate run is a Java method and its pre- and postconditions (in JML). Test cases are automatically generated for the method using a tool called EvoSuite [8]. The Daikon tool, which is described in detail in the previous section, will automatically generate candidate loop invariants based on the execution of the generated test cases. Candidate loop invariants are then verified with ESC/Java2. If the verified loop invariants are sufficient to prove the method the algorithm is done. Otherwise more test cases are generated and Daikon is run again. This can happen for several iterations until either proving the method succeeds or until no more valid loop invariants are found by Daikon. When the latter happens the GIN-

DYN tool is used. This tool creates candidate loop invariants by generating mutations of the method postcondition. The set of candidates is pruned by rejecting those that fail at least one test case, and by eliminating tautologies. ESC/Java2 is then used to verify the remaining candidates. This process is also repeated until verification of the entire method succeeds. If the entire process takes too long the program gives up and reports a failure.

With an average run time of about 45 minutes per method DynaMate is not very fast. The authors suggest this run-time might be improved by not treating the third-party components as black boxes.

The underlying tools use randomness in their operation so for some methods it will only be able to prove them on some runs. For example, the authors report that over 30 runs DynaMate was able to prove `ArrayList.lastIndexOf` correct only 20% of the time.

3.4.1 EvoSuite

EvoSuite is a tool that can automatically generate test suites for Java programs. The tool will attempt to evolve a small test suite with high code coverage. It achieves this by automatically generating multiple test suites, and then uses a genetic algorithm to evolve a better generation of test suites. Fitness of a particular test suite is calculated based on branch coverage. The test suites created by EvoSuite are useful when performing dynamic program invariant detection with tools such as Daikon because their high branch coverage will ensure that most corner cases are observed.

3.4.2 GIN-DYN

The GIN-DYN tool attempts to guess loop invariants based on the postconditions of a method. The tool is based on `gin-pink` [9], which is discussed in detail in the previous section. However, it operates slightly differently. Firstly it operates on a single loop at a time, rather than generating mutants using the entire method body and then trying to use them as loop invariants for each loop. All clauses occurring in the method postconditions are used as initial mutants. GIN-DYN also uses different mutation operators, which we will describe below. The authors provide no data to suggest that their mutation operators are better than those used by *gin-pink*, but apart from the *aging* mutation they appear to be at least as powerful as the operators used in *gin-pink*.

First there is *substitution*, where a subexpression in a mutant is replaced by an expression of the same type that occurs somewhere in the current loop. This mutation strategy can do everything that the *constant relaxation* and *uncoupling* mutations from gin-pink can do and more, because the operators in *gin-pink* can only operate on constants and variables rather than expressions.

Secondly we have the *weakening* mutation, which takes a mutant and a boolean expression occurring in the loop body and generates two new mutants, one where the original mutant is implied by the boolean expression, and one where it is implied by the negation of the expression. For instance, if our original mutant is m and b is a boolean expression, then the new mutants are $b \implies m$ and $\neg b \implies m$.

Finally GIN-DYN has the *aging* strategy, which is a much simpler version of the mutation strategy with the same name that is used in gin-pink. The GIN-DYN aging mutation simply takes a mutation containing a subexpression e of type **int**, and generates two new mutants, where e is replaced by $e - 1$ and $e + 1$ respectively.

Because randomly applying these operators leads to a combinatorial explosion, GIN-DYN has some predefined sequences of mutations called *waves*. These are sequences that are likely to yield good results. After running a wave, bad mutants are rejected by checking if they hold at the start and the end of the loop body when running the test cases generated by EvoSuite. Any hypothesized invariant that does not hold in at least one of the tests is obviously invalid and is removed. Note that all generated mutants can be tested at once in a single execution of the test suite. Some of the mutants that remain may be redundant because they are implied by loop invariants that have already been found. These mutants are also removed by generating for each mutant a dummy method that contains only all confirmed loop invariants as an assumption, and the mutant as an assertion. If the method is verified by ESC/Java2 then the mutant must be an implication of the invariants that are already known, so it can be removed. Finally the remaining mutants are added to the actual method and tested for validity.

3.5 Summary

In this chapter we have described several loop invariant generation techniques. Many of these use static analysis, while others will attempt to derive loop invariants based on observed program executions. The work by Galeotti et al. on DynaMate serves as an inspiration for the loop invariant generation framework we have created. The other techniques are good examples of additional improvements that could be added to the framework as possible future work. Because of the varied nature of the the loop invariant generation techniques that currently exist, and because of the success of DynaMate even with only a very limited selection of loop invariant generating tools, we expect a framework that incorporates more of these techniques to be very effective.

Chapter 4

Setting up the Case Study

As described in the introduction, we will create a framework that integrates multiple different loop invariant generation tools. The creation of this framework will serve as a case study to identify technical difficulties that may be blocking the integration of formal methods tools in general. In this chapter we will explicitly list our goals, and formulate a list of requirements for the framework based on these goals. These requirements will guide the design of the framework

4.1 Goal

The work by Galeotti et al. [10] shows that multiple different tools can complement one another, and this combination of tools is able to perform better than any individual tool is able to. Their research focused primarily on demonstrating the synergy between their chosen tools, but they did not report on the difficulties of integrating these different tools within one framework. The goal of our study is to investigate technical difficulties that stand in the way of integrating formal methods tools in other software.

Even though we will add a limited number of tools to our framework, we will set it up in such a way that new tools can be added by writing some glue code that sits between a new tool and the framework. This setup will force a design that isn't specialized for the types of tools we use. This is desirable because a design that is too focused on the particular tools we use may hide difficulties that would appear when writing a more general framework.

The problems we encountered while implementing the framework are documented in chapter 7. Based on these problems we will formulate advice for tool creators that may help improve the easy with which their tools can be integrated in the future.

4.2 Requirements

Using the goals discussed in the previous chapter we have formulated requirements for our framework. These requirements are listed below, and each individual requirement will be discussed in detail in the subsections following this list.

1. The framework should integrate several different tools.
2. The framework should reuse existing functionality where possible.
3. The framework should be extensible

4.2.1 The framework should integrate several different tools

We believe loop invariant generation tools make a good target for integration because there are many different ones with various strategies, and they can complement one another as shown by Galeotti et al. [10]. Because these tools are created by different researchers, we believe this diversity will contribute to discovering more difficulties that hinder integration.

4.2.2 The framework should reuse existing functionality whenever possible

If tools were to use the same software libraries where possible they might be easier to integrate. This would not apply to internal details, but could make a big difference for types exposed through an API. For instance, if two tools operating on Java were to use different libraries for interacting with Java code, there might be additional friction when integrating these two tools.

4.2.3 The framework should be extensible

While it may be possible and easier to build a piece of software that handles only the specific tools that we have chosen to integrate, this would likely not reveal the difficulties that would arise when integrating arbitrary tools. Therefore we will set up our framework in a generic way such that multiple tools can be added, and use those generic ways to integrate with the tools we have chosen.

4.3 Conclusion

We will use EvoSuite for test-case generation, and Daikon for dynamic invariant detection. We know from DynaMate that these two tools work well together. In additional to these tools we will write a simple static analysis technique which will perform static invariant generation. We will use OpenJML to tie everything together: the framework will use OpenJML to parse Java code, analyze Java code, and, where required, to modify Java code. In addition, OpenJML will be used to verify generated invariants.

One advantage of the chosen tools is that they are all written in Java. We believe that a shared implementation language will maximize the possibility for integration, because there are no extra language barriers. Writing a static analysis pass using OpenJML will test the viability of using OpenJML as a basis for writing static analysis tools.

Chapter 5

Architecture

Interoperability between different tools often requires that the results of one tool can be used as the input for another. For instance, when a tool generates invariants that are suspected, but not proven to be true, we would like to pass these suspected invariants to to tools that can prove (or disprove) invariants. None of the tools we use are directly usable in this way. To get these tools to interoperate we introduce an intermediate format, and will introduce functionality to translate between this intermediate format and the inputs and outputs of the various tools.

Even with translation of inputs and outputs not all tools need to directly interact. A tool that generates testcases does not produce output that is going to be of use for a static analysis tool. We have listed the invariant generation paths that we intend to support in figure 5.1. The tools we will use will all fall into one of these categories. In the next section we will discuss these categories in more detail.

In addition to keeping track of the various tools that it can run, the framework should keep track of the loops it is attempting to prove and of the invariants that have been found. Because some invariants are merely suspected and not (yet) proven, the framework has to keep track of the status of each invariant as well. It may be impossible to prove a particular suspected invariant until other invariants have been found, so suspected invariants cannot simply be discarded when proving it fails the first time (although naturally a counter-example is always sufficient to reject it immediately).

The framework will keep track of the loops by holding a representation of them in memory.

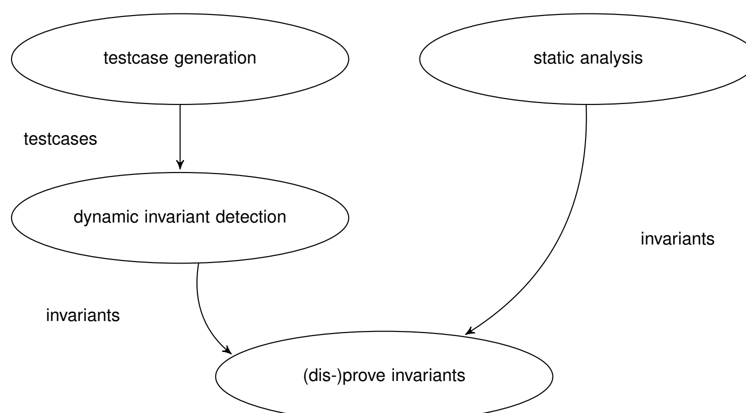

Figure 5.1: Invariant generation

For each loop the framework will also keep track of the variables that are in scope. This information will be exposed to the code that runs the tools, which can use it to translate tool results to the intermediate format.

5.1 Framework design

The framework we come up with consists of several parts. We first observed that there are different types of tools among the ones we will use: tools that generate test-cases, tools that perform dynamic invariant detection, tools that perform static invariant detection ("static analysis"), and tools that can verify or reject suspected invariants. In the following subsections we will discuss each of these types in turn.

5.1.1 Test-case generation

Tools that dynamically detect invariants need to observe program executions to collect information. Ideally every program would have a good test-suite that can be used for this purpose, but when this is not the case it is possible to automatically generate test-cases. A good test-case generator can also supplement an existing test-suite. Tools such as EvoSuite will attempt to generate test-cases that have full code coverage, which is good for dynamic invariant detection tools because it allows dynamic invariant detection over not just the common paths but also the edge cases.

In our framework a test-case generator can be requested to generate executions for some classes. Because there is no practical limit to the number of test-cases that can be generated, the test-case generator may decide for itself when it has produced enough tests. The framework can request more executions when the ones that were generated previously turn out to be insufficient.

5.1.2 Dynamic invariant detection

Tools such as Daikon are able to record the program state at various points in the execution of a program, and use this recorded information to make informed guesses about invariants of that program, based on the invariants that hold in the observed executions. By observing a large and diverse set of executions the invariants observed by a dynamic invariant detector are more likely to match the true invariants of the program.

The framework will support tools that perform dynamic invariant detection. These tools will receive their program executions from the test-case generation tools. Because invariants found through dynamic invariant detection are always a best guess based upon observations, invariants found by these tools will always need to be verified by other tools.

Dynamic invariant detection can discover invariants such as $result \geq 0$ in the code example 5.2, even when no information about *Math.max* is available. Naturally any tool attempting to prove the correctness of this invariant would still require access to additional information about *Math.max*.

```java
    public static int example(int[] arr) {
        int result = 0;
        // Can be detected by a dynamic invariant detector:
        //@ loop_invariant result >= 0;
        for(int i : arr) {
            result = Math.max(result, result + i);
        }

        return result;
    }
```

Figure 5.2: Dynamic invariant detection example

5.1.3 Static analyzers

Tools performing static analysis represent the classical approach to proving program correctness. While discovering all relevant loop invariants using only static analysis is impossible in the general case, there are many cases where static analysis is able to generate some or all loop invariants necessary to prove correctness of a method. A good example of such a tool could be Valigator, which analyses the relations between numbers in a loop.

Our framework will not require a static analysis tool to be sound, a static analyzer may indicate for the invariants that it has found whether they should be considered proven or not. Unproven ("hypothesized") invariants will be checked by other tools. Invariants reported as proven will be assumed to be correct by the framework. The GIN-DYN tool used by Dynamate is a good example of an unsound static analysis tool: it will guess invariants by mutating the method post-conditions, and most of the invariants that it produces will be incorrect.

We will add a static analysis pass of our own to the framework which will guess invariants for *for*-loops. In our experience, many *for*-loops have the following shape: *for (int i = a; i < b; i++) { /* loop body */ }*, where *a* and *b* are expressions, and *int i* can actually be of any numeric type, and have any name. A simple static analyzer that ignores the loop body and simply guesses that *i >= a* and *a <= b* will often be correct. Our implementation will report the invariants it generates as hypothesized. Tools such as OpenJML can usually trivially prove these invariants when they are correct.

5.1.4 Invariant checkers

Some of the tools mentioned above produce invariants that may be incorrect. To make certain the framework does not report incorrect invariants, we can use tools that can distinguish between correct and incorrect invariants. We will use OpenJML to make this distinction. By using static analysis it can attempt to prove or disprove hypothesized invariants. If the invariant checkers are unable to either prove or disprove an invariant it will be remembered so that additional attempts to prove or disprove it can be made once more correct invariants have been found.

5.2 Components

To keep the framework extensible, we will define interfaces for each tool-type discussed in the previous section. Adding a new tool can then be achieved by implementing the appropriate interface. We refer to the combination of a tool and the wrapping code as a component, and the interfaces all inherit from a base interface called `Component`.

The `Component` interface defines an `init` method which is called once for each component to initialize it. We observed that many tools needed to interact with the filesystem in some way, so the `init` method has a path to a folder where the tool is allowed to create files and folders. By ensuring that all tools have their own folder, there is no risk of tools overwriting files of other tools. In addition to a path, the init method has a `Context` parameter which is an object containing various methods for obtaining information about the Java code that the tools must find loop invariants for. This information includes which classes are targeted for loop invariant generation, and information about the loops in each class. This information includes a reference to the abstract syntax tree node representing the loop, a list of all the variables that are in scope for the loop, and a list of all variables that are used in the body of the loop.

In addition to the `init` method, `Component` defines a method `getName`, which returns the name of the underlying tool. This is used for diagnostic messages.

5.2.1 ExecutionProvider

The `ExecutionProvider` interface extends the `Component` interface. It has a method `provideExecutionsFor`, which takes a list of Java class names as a parameter, and returns a list of `Execution` instances. The `Execution` interface represents a program execution that can be used by dynamic invariant generators to run code which triggers executions for the requested classes. EvoSuite uses this interface and the instances it returns contain information about how to run the generated unit-tests.

5.2.2 DynamicInvariantGenerator

The `DynamicInvariantGenerator` interface is implemented by tools that can dynamically detect (likely) loop invariants, such as Daikon. The interface extends `Component` and adds a single method `generateInvariants`, which accepts a list of `Execution` instances, and returns loop invariants. Because dynamic invariant detection can only detect "likely" invariants, implementations of this interface will generally mark the resulting invariants as "hypothesized" , leaving the work of proving or rejecting the invariants to other tools.

5.2.3 StaticInvariantGenerator

Tools using static analysis to generate loop invariants can use the `StaticInvariantGenerator` interface. This interface has a single method `generateInvariants`, which has no inputs and returns invariants. While most static analysis tool will not return different results when invoked

multiple times, this is not guaranteed. The GIN-DYN tool used in DynaMate generates possible invariants by randomly mutating method post-conditions. Therefore tools of this type will also get executed for each iteration of the framework, and have the option of marking reported invariants as “proven” (for sound static analysis tools) as well as “hypothesized” (for tools such as GIN-DYN).

5.2.4 InvariantReject

The `InvariantReject` interface is intended for tools which can reject incorrect invariants, as well as those which can prove the correctness of correct invariants. It has two methods. The first method is `assumeInvariants`, which takes invariants that have already been proven as an argument, and tells the tool to assume that these invariants are correct. This can be useful when a different tool has proven correctness of one or more invariants. The second method is `rejectAll`, which takes a collection of invariants and should attempt to prove or reject these invariants, and return three sets of results (proven, hypothesized, and rejected). The hypothesized state is for invariants that the tool was unable to conclusively prove or disprove, these invariants may be resubmitted once additional invariants have been proven.

5.3 Operation of the framework

The workings of the framework itself are simple. The framework will load and initialize the various components. Then it will enter a loop, where it will run the different component types in turn, and collect sets of confirmed, hypothesized, and rejected invariants. At the end of each iteration, it compares these sets with those found in the previous round, and when there are no changes it exits the loop. Note that this does not imply that no more invariant could have been found; for instance, dynamic invariant detection is limited by the executions it has observed, so generating more test-cases might have enabled the detection of more invariants.

The framework will load the various components and initialize them. After that it will enter a loop where it will run the components. After each iteration it will check if it progress has been made. The framework measures defines progress as any mutation to the sets of proven, hypothesized, and rejected invariants. When no more progress is being made the framework will exit the loop and shut down.

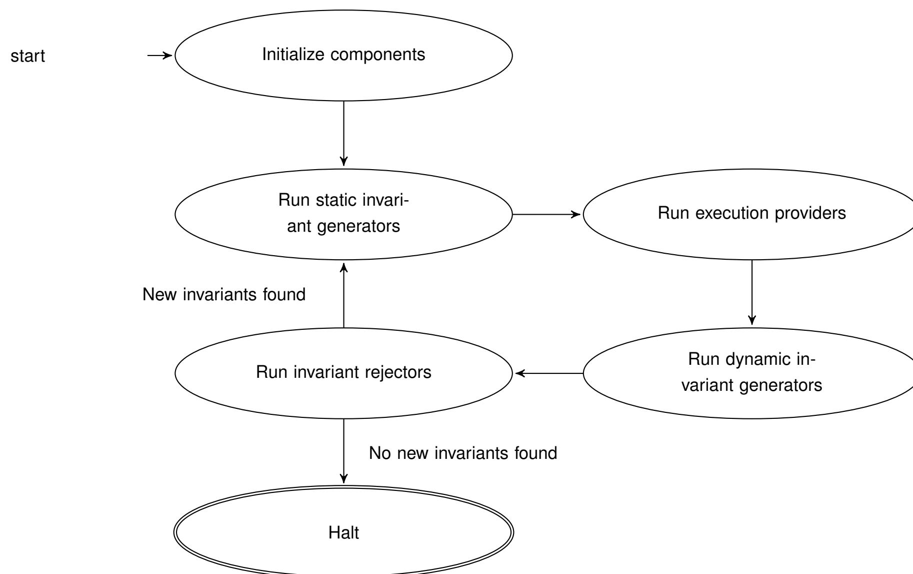

Figure 5.3: Main framework loop

Chapter 6

Implementation

The part of the framework which initializes all the components and runs the main loop is straightforward. It has a hard-coded list of components that are constructed and initialized. After that it will initialize a new instance of the `LoopRunner` class, which will then enter the main loop and run the components in turn until no new invariants are found, as described in the previous chapter. The implementation complexity resides in two parts: the analysis of the Java files that the framework operates on, and the implementation of the various components.

In this chapter we will first describe how the context is created, and then describe each component we implemented for the framework.

6.1 Building the Context

For analyzing the Java files the framework will initialize a `Context` object. OpenJML has an API that enables using it as a library, and the `Context` object will create an instance of OpenJML inside the current process and expose its API. The `Context` object will load the Java files in OpenJML and run an analysis to find all loops in the given files. For each loop, it will store:

1. The variables that are in scope for that loop
2. The variables that are defined within the scope of that loop
3. The variables that are used in that loop
4. The method that contains the loop
5. The class that contains the loop

The context will be passed to all components, which can use the gathered data. Gathering the above information is done in two passes over the abstract syntax tree built by OpenJML, and most of the components we created use this information in some way. Having this information in a context object available to each component avoids having to re-implement the functionality in each component.

6.2 EvoSuite

The framework uses EvoSuite for its test-case generation. During each iteration of the main loop, the framework will have EvoSuite generate a new test-suite. While it is possible for EvoSuite to mutate an existing test-suite, we have chosen not to use this functionality, but to generate an entirely new suite each time. This choice avoids having to distinguish between the case of having to generate a new test-suite and the case of having to mutate an existing one.

EvoSuite can be set to keep working until it has achieved 100% branch coverage, but such coverage is not possible for all possible inputs, nor is it possible to accurately predict in advance how much branch coverage can be achieved in the general case. To ensure that EvoSuite terminates, we set a time limit. The framework starts this time limit at 10 seconds, and will exponentially increase this limit each iteration.

6.3 Basic static invariant generator

We made the observation that many for-loops behave very predictably: they assign an initial value to a single variable, and that variable is then only ever incremented or decremented, usually monotonically. It can take a couple of iterations before dynamic invariant detection gains enough confidence to emit these properties, especially since for-loops often iterate over the indices in some list or array, and EvoSuite tends to generate relatively short inputs. To work around this we have created an extremely simple static invariant generator, which visits every for-loop and hypothesizes these invariants. This guessing of invariants is safe to do because invariants will be checked with OpenJML, which will usually instantly either reject on confirm the correctness of these invariants.

6.4 Daikon

When other components have generated testcases, the Daikon component can run Daikon to perform dynamic invariant detection on testcase executions. However, this involves more than simply running the tool. Daikon will take snapshots of variables when entering and leaving a function, but it will not detect any information about loop executions. To extract information about loop executions, we must modify the source to insert dummy function calls. In addition, even though Daikon can report the invariants that it has found in JML syntax, this JML is not directly usable and requires a translation. In the following paragraphs we will explain how Daikon is used in the framework.

6.4.1 Gathering loop information

Using the data that the `Context` object provides, the Daikon component knows which loops it must investigate, and which variables are in scope and used in these loops. It will generate dummy classes containing dummy methods, and insert calls to these methods in loop bodies. These operations are performed on the abstract syntax tree provided by OpenJML. An example

```
/* Original method */
void example(int[] args) {
    int sum = 0;
    for (var i = 0; i < args.length; i++) {
        sum += args[i];
    }
}

//////////////////////////////
/* Modified method */
void example(int[] args) {
    int sum = 0;
    for (var i = 0; i < args.length; i++) {
        try {
            dummy_method(args, sum, i);
            sum += args[i];
        } finally {
            dummy_method(args, sum, i);
        }
    }
}

/* In a dummy class: */
void example_0(int[] args, int sum, int i) {}
```

Figure 6.1: Daikon loop transformation

of this transformation is shown in Figure 6.1. The original abstract syntax tree is left intact by copying it and applying these transformations on the fly.

After we have obtained the transformed methods and dummy classes, these are all written to disk and recompiled. We then run Daikon on the testcases, which will execute our instrumented classes. Daikon will record calls to the dummy methods in the loop bodies. When the test-case execution is complete, we can ask the tool to generate invariants based on the gathered information. The tool will then emit invariants that held over all observed executions.

6.4.2 Interpreting the results

After Daikon has run on the testcases we need to interpret the results. Daikon has the ability to run on multiple trace files, and we include not just the trace of the most recent test executions, but also all those that have come before. This way Daikon has more data from which it can attempt to extract invariants. This is important because dynamic invariant detection always carries the risk of false positives. Daikon manages to be useful despite the risk of false positives by not reporting found invariants that are likely to arise “by chance”. For instance, if Daikon were to only observe a single call of a method, then based on that data it might be justified to emit the invariant that the parameters of that method are always the ones passed in that execution. In practice Daikon will suppress reporting many invariants until it has gathered enough data that, given a set of reasonably representative observed executions, the invariant is unlikely to arise by chance.

The detected invariants are reported per method, for both the method entry as well as the

method exit. Because we are only interested in the invariants detected on our dummy methods, and these dummy methods have empty bodies, we can afford to only look at the invariants that are found on method entry. The invariants on method exit will be exactly the same, but will also include statements such as `\old(param) == param`. By ignoring the exit invariants we do not have to filter out such tautological statements.

By remembering an association between the loop that we are interested in and the names of the generated dummy methods, we know which loops the detected invariants must apply to. In addition we reuse the original variable names as the names of the dummy method paramaters, thereby ensuring that translation is easier.

Daikon has the option to report its finding as JML. Sadly these invariants are not directly usable because they often contain references to the `daikon.Quant` class. For instance, let `a` and `b` be variables, where `b` is a collection that contains `a`. Daikon will report this invariant as `daikon.Quant.memberOf(a, b)`. OpenJML does not know what this means, so this must be translated to "normal" JML. Our Daikon component contains transformations to do this.

The static members on `daikon.Quant` that Daikon uses to represent invariants cannot be trivially translated to JML. For instance, the `daikon.Quant.memberOf` method mentioned above has 18 overloads, all of which have two arguments, and the emitted invariants do not indicate which overload was chosen. For instance, `daikon.Quant.memberOf(a, b)` might mean `\exists int i; 0 <= i && i < b.length; b[i] == a` if 'b' is an of an array type, but if b is a `List`, then the meaning would be `\exists int i; 0 <= i && i < b.size(); b.get(i) == a`.

To translate the references to `daikon.Quant` into ordinary JML we first use OpenJML to parse the expressions. The framework has the entire type-checked source abstract syntax tree in memory, and we combine this information with the parsed expressions to determine how to perform the translation. Using this technique we are able to translate most invariants emitted by Daikon into ordinary JML, removing all references to the methods on `daikon.Quant`. These translated invariants can then be reported to the framework, which can then attempt to prove or reject them.

6.5 OpenJML

We use OpenJML to test invariants that are not yet proven or rejected. This is done by making a copy of the original abstract syntax tree, and while copying inserting the suspected loop invariant. Proving one loop invariant may depend on another, so in addition to the suspected loop invariant we insert all other loop invariants that are known to hold for that loop. Then we let OpenJML attempt to prove the method. When OpenJML succeeds, we know the invariant to be correct. When OpenJML indicates that it has found a counter-example, we know the invariant can be rejected. It is also possible for OpenJML to indicate that proving was inconclusive. When this happens the loop invariant will keep its 'suspected' status and we will try to prove it again when additional other loop invariants have been found.

Chapter 7

Findings

During implementation we ran into some bugs and challenges with the tools involved. We will discuss the limitations that concern the suitability of OpenJML as a library for interacting with Java code, as well as those that we believe to be more likely to arise when integrating different tools. We will not discuss individual bugs that are restricted to a single tool and are not triggered or exacerbated by being included in a framework.

7.1 Documentation

One of the biggest problems we encountered was not caused by the implementation of the tools, but by limited technical documentation. Due to this lack of documentation we ended up at several dead ends during the implementation of the framework, trying to accomplish things in a certain way that could not work for reasons that were not obvious until a lot of time had been wasted doing something incorrectly.

Of the various tools we interacted with OpenJML the most, and while we do not believe the documentation is OpenJML is worse than that of the other tools, it was due to this tight integration that missing technical documentation was most noticeable. For example, the user-guide of OpenJML contains a separate chapter about using OpenJML and OpenJDK within other programs, but this chapter turned out to be far from complete. It contains some very useful sections about creating a compilation context and parsing Java code, which make it easy to get started. However, the chapter contains various sections that do not yet have any content, such as the ones about type-checking, static checking, and modifying abstract syntax trees. We had to discover how to perform these actions through trial and error.

After learning more about the workings of OpenJML it was relatively easy to work with, but the lack of technical documentation may well be preventing it from being integrated in other programs.

We found that the EvoSuite and Daikon tools have good documentation on how to use these tools, and even some documentation for modifying and extending them. However we were unable to find any documentation on how to integrate either tool in another program, and resorted to just creating a new process and passing the appropriate command line options. Later we discovered that EvoSuite can leave threads alive in the background, and Daikon will

create a new process with a custom class-loader as part of its operation anyway. For these reasons we believe that creating a new process for these two tools instead of attempting to run them inside the process of the framework was the best outcome.

7.2 Tools may not clean up properly after running

The EvoSuite tool has a Java API, suggesting that it can be used without creating a new process. However, it will create new threads during test-case generation, which may be left running even after test-case generation is complete. This can keep the framework process alive even after the main thread is done. The EvoSuite tool deals with this by explicitly calling `System.exit(0)` at the end of its main method, but this is not a solution when integrating EvoSuite in another program.

In our framework we wish to invoke EvoSuite multiple times, but we do not wish to accumulate an ever-growing number of unnecessary background threads. To solve this we must launch a new process whenever we wish to run EvoSuite. While this works well, this solution clearly limits integration: only the normal command-line interface can be used, and the Java API cannot be used. We hope this threading issue will get fixed so that better integration becomes possible.

7.3 Modifying the AST in OpenJML

In our framework OpenJML is used as much as possible for operations interacting with the input source code. For instance, for dynamic loop invariant detection with Daikon it is necessary to insert new function calls around loop bodies so that Daikon knows how to make snapshots of the variables. We perform these operations by copying the abstract syntax tree created from the original source code and injecting the new parts on the fly. We originally tried to then continue compilation with our modified code, but this does not work: we needed to perform type-checking to get the instrumentation right, and OpenJML doesn't support repeating type-checking. In the end we had to resort to writing out our modified source code to files, and shell out to `javac` to compile these files. We were unable to find a way to get this to work inside OpenJML without creating a new process.

For OpenJML to be a better library to use for modifying abstract syntax trees it should be possible to repeat type-checking after modifications have been made. We suspect the simplest way to support this use-case would be for OpenJML to discard any previous type-checking results when type-checking is requested. Naturally this would lead to unnecessary work being performed, because a class that hasn't itself been modified and does not reference any modified class will still type-check exactly the same, but it would still be much more efficient and ergonomic than the current workaround.

During development some other issues with OpenJML were found as well, mostly related to incorrect output when converting abstract syntax trees back to strings. These issues were fairly minor and quickly fixed after being reported to the project. We do not believe that such issues would block adoption of OpenJML as a commonly used library to interact with Java

code, as they were easy to work around, and if more exist they would very quickly be found (and fixed) when more people start using OpenJML.

7.4 Interactions between EvoSuite and Daikon

While developing our framework we ran into an unexpected interaction between EvoSuite, Daikon, and the Java standard library. The interaction between these components caused a problem that was highly unlikely to occur when these components were used in isolation.

1. EvoSuite will often use the `List<E>.subList(int, int)` method on an existing list to create additional lists. It is not clear to us whether it does this intentionally to increase code coverage.
2. Java specifies that the semantics of the list returned by `subList` are undefined when the original list is structurally modified. In practice, the Java standard library will attempt to detect this occurrence and throw an exception to warn the developer.
3. When checking invariants, Daikon will call methods such as `List.toArray()` on parameters that happen to be a list.

Individually none of these things would be problematic. When combining them, it turned out that Daikon would unintentionally trigger a `ConcurrentModificationException` when attempting to get the values in the lists, because EvoSuite would create lists that were either outright invalid (in the sense that performing any operation on them would be undefined and trigger an exception) or would cause other lists to become invalid after the first modification.

In Figure 7.1 we demonstrate a small program that runs without errors when executed normally. However, when running this program with Daikon an exception will be thrown. The exact steps that trigger this are as follows:

1. The parameter `bomb` is a sublist of parameter `trigger`.
2. Modifying `trigger` invalidates `bomb`, because sublists are invalidated when the backing list is modified.
3. When exiting the `detonate` method, Daikon will record the values of the parameters.
4. The sublist `bomb` throws a `ConcurrentModificationException` when `toArray` is invoked.

While code such as in the given example program is unlikely to be written by a human, EvoSuite will use `subList` in generated unit tests when creating inputs, thereby triggering these exceptions.

This problem has since been solved in Daikon, but we think it is a good example of unexpected problems that can occur when combining different tools.

```java
public static void main(String[] args) {
    java.util.List<Object> base = new java.util.ArrayList<>();
    detonate(base.subList(0, 0), base);
}
public static void detonate(
        java.util.List<Object> bomb,
        java.util.List<Object> trigger) {
    trigger.add(new Object());
}
```

Figure 7.1: Code example that crashes Daikon

Chapter 8

Advice

Based on our experiences building the framework we have formulated several recommendations, that we will discuss below. For each recommendation we will discuss the impact we think it would make, and the effort required on the part of tool builders.

8.1 Build tools as a library

The tools that we used were clearly intended to be used from the command line. While such a tool can still be integrated in other software by creating a new process with the correct parameters, running it, and then parsing strings to interpret the results, this does form a barrier for integration. If tools were to present an API, they could be used as a library by other programs. This has two big advantages:

The first advantage is in argument passing. A command-line interface allows only strings to be passed as arguments. While this limitation does not pose a problem for command-line programs, it can raise additional barriers for programs trying to integrate with a tool. For example, some tools take one or more files as inputs. A command-line program will usually accept arguments that represent a path to these files. A program integrating a tool may wish for that tool to operate on data that exists only in memory. When a tool accepts its inputs only as file-paths the program wishing to use that tool is forced to create temporary directories, write the data to a file, run the tool with a path to the file, and possibly ensure that the temporary files and directories are removed. None of this would be necessary if the tool provided an API that accepts streams.

The second advantage is in returning results. A command-line tool will usually report its results by outputting text. A program wishing to use this output will be forced to convert these strings back into something it can understand. An API would allow the tool to return strongly typed objects. A program using this API can then choose to operate on these objects directly, or possibly convert them to a different format. Either way avoids the need to parse text, which greatly simplifies interoperability.

Of the tools that we have used we think OpenJML exemplifies this approach best. All of the functionality we needed could be accessed by loading it as a library and using the API. The API has many convenient methods that allow loading Java source code not only from files on

a disk, but also from a string or an arbitrary stream.

One additional important restriction is that the functionality in the “library”-part of a tool should avoid performing actions that negatively affect the current process, such as ending the process. If the functionality behind an API will sometimes deliberately end the current process, then a program integrating with that API will still be forced to create a new process to interact with that tool.

8.2 Have technical documentation on using the API

As mentioned in the previous chapter, missing documentation about using the OpenJML API cost us a lot of time. For Daikon and EvoSuite we could not even find documentation about integrating either of them in another program. If a tool has an API that could be used for integration, but is missing documentation on how to use this API, then that API will probably not get used. We would recommend tool creators to document the usage of their API. If a program has various public classes and methods but is *not* intended to be integrated directly in another program, then documenting this limitation can also be valuable.

8.3 Avoid global state

The use of global state is often tempting when developing a small tool, because it can avoid the need to have a context object instance that would otherwise be required at many places in the program. However, when the tool is made part of a larger program, the global state can become a liability. The global state may prevent the functionality of the tool from being invoked multiple times or from using the tool in a multi-threaded environment.

Daikon is one tool which could use this advice. The main `Daikon` class is never constructed, and keeps track of all of its state in static variables. Clearly this design choice makes it impossible to use this class from multiple threads at the same time, because there would be many opportunities for race-conditions. Perhaps the designers of Daikon have a good reason for this design, but it does introduce extra complexity when integrating with other programs.

8.4 Use common libraries

Tools tend to operate on objects in some domain. For example, in our case-study the tools we used operated on Java programs, either as source-code on as binaries, and produced invariants. Quite a lot of time had to be spent on translating these invariants into a format that the framework could work with, and finding the particular loop that it applies to. We believe the use of common libraries in program APIs would facilitate tool interoperability by ensuring that fewer translations between different formats are required.

One great example of a common library facilitating interoperability among formal methods tools is SMT-LIB [2]. It defines a common input and output language for SMT solvers, making it

very easy to swap one SMT solver for another. OpenJML uses the format defined by SMT-LIB, enabling it to easily support several different SMT solvers.

8.5 Future work

We see various avenues for future work. Below we will list some of these avenues.

8.5.1 Investigate common needs in research tool development

Interoperability between different tools is easier when they are more alike. Tools within a common domain will likely have several actions in common that could be performed by a common library. For example, when generating loop invariants for Java programs common actions include parsing and interacting with Java code (especially for tools performing static analysis) and reporting detected invariants. Future work could include investigating various categories of formal methods tools and checking which operations are shared between tools within a certain category. It would be interesting to see which shared libraries are used, and if there is a need for additional libraries.

8.5.2 Further research into combining the results of multiple loop invariant generation tools

The work by Galeotti et al. on DynaMate has demonstrated that combining multiple different tools for loop invariant generation can be more effective than using these tools individually. Does adding additional tools produce even better result? Which tools synergize best?

Originally we had intended to include Valigator by Kovács et al. [11], but this tool does not accept Java as an input language. Translating Java to their input language turned out to be hard, and therefore we did not include this tool in our framework. However, because Valigator is very different from the other tools we have used, we did expect it to supplement them well. For this future work we would recommend taking a number of dissimilar tools and using them together on some test input. Comparing with various subsets of the chosen tools might give information about which tools complement one another best.

8.5.3 Investigate combining tools in other areas in formal methods

Fully automatic loop invariant generation is an interesting subject because it is both clearly beneficial and impossible to do in the general case. These properties have resulted in a large number of techniques which are able to perform loop invariant generation in specific cases. Investigate whether there are other areas in the field of formal methods that have similar conditions, and check whether combining different techniques also yields good results.

8.6 Conclusion

After performing our case study we believe that the formal methods we have experimented with still have a lot of room for improvement when it comes to ease of integration. When we started we were expecting that integration would be much easier that it turned out to be. Based on this case-study we have a strong impression that formal methods tools are not made with integration in mind.

We were positively surprised by the quality of the usage information for all of the tools we have used. It seems clear to us that the people behind these tools do want others to use them, and are willing to invest time in making this possible. Based on this we hope that ease of integration will also get priority when more research is performed into the effectiveness combining complementary tools.

We hope that tool creators will take our advice to heart, and that tools created in the future will be easier to integrate in other software. While combining existing tools is less exciting that creating entirely new algorithms and techniques, we believe it to be important work, that will increase the effectiveness of formal methods, and hopefully lead to increased industry use of formal methods. Making tools easier to integrate in existing software has the additional effect of making it more attractive to integrate in integrated development environments, and hopefully this is a place where many formal methods tools will ultimately end up.

Bibliography

[1] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *Computer aided verification*, pages 171–177. Springer, 2011.

[2] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. The smt-lib standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, volume 13, page 14, 2010.

[3] David R Cok. Openjml: Software verification for java 7 using jml, openjdk, and eclipse. *arXiv preprint arXiv:1404.6608*, 2014.

[4] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[5] Bruno Dutertre and Leonardo De Moura. The yices smt solver. *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, 2(2), 2006.

[6] Michael D Ernst, Jake Cockrell, William G Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *Software Engineering, IEEE Transactions on*, 27(2):99–123, 2001.

[7] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):35–45, 2007.

[8] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419. ACM, 2011.

[9] Carlo Alberto Furia and Bertrand Meyer. Inferring loop invariants using postconditions. In *Fields of Logic and Computation*, pages 277–300. Springer, 2010.

[10] Juan P Galeotti, Carlo A Furia, Eva May, Gordon Fraser, and Andreas Zeller. Automating full functional verification of programs with loops. *arXiv preprint arXiv:1407.5286*, 2014.

[11] Laura Kovács. Automated polynomial invariant generation by algebraic techniques for imperative program verification in theorema. *Giese and Jebelan (2007) pp*, pages 56–69, 2007.

[12] Laura Kovács and Tudor Jebelean. Finding polynomial invariants for imperative loops in the theorema system. In *Proc. IJCAR 06 Workshop Verify 06*, pages 52–67. Citeseer, 2006.

[13] Gary T Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, Daniel M Zimmerman, Werner Dietl, et al. Jml reference manual, 2013. Revision 2344.

[14] Stephen Magill, Aleksandar Nanevski, Edmund Clarke, and Peter Lee. Inferring invariants in separation logic for imperative list-processing programs. *SPACE*, 1(1):5–7, 2006.