Advancing Practical Specification Techniques for Modern Software Systems

John L. Singleton

CS-TR-18-01

Revised April 16, 2018

Keywords: Deductive verification, Java Modeling Language, specification inference, verification, formal methods, predicate transformers, Strongarm, Spekl, Evidently, information flow

2013 CR Categories: D.2.4 [*Software Engineering*] Software/Program Verification — Formal methods, programming by contract; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — Assertions, logics of programs, pre- and post-conditions, specification techniques;

Ph.D. dissertation.

Computer Science

4000 Central Florida Blvd.

University of Central Florida

Orlando, Florida 32816, USA

ADVANCING PRACTICAL SPECIFICATION TECHNIQUES FOR MODERN SOFTWARE SYSTEMS

by

JOHN L. SINGLETON
B.S. University of Central Florida, 2014
M.S. University of Central Florida, 2016

A dissertation submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy
in the Department of Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Spring Term
2018

Major Professor: Gary T. Leavens

ABSTRACT

The pervasive nature of software (and the tendency for it to contain errors) has long been a concern of theoretical computer scientists. Many investigators have endeavored to produce theories, tools, and techniques for verifying the behavior of software systems. One of the most promising lines of research is that of *formal specification*, which is a subset of the larger field of *formal methods*. In formal specification, one composes a precise mathematical description of a software system and uses tools and techniques to ensure that the software that has been written conforms to this specification. Examples of such systems are Z notation, the Java Modeling Language, and many others. However, a fundamental problem that plagues this line of research is that the specifications themselves are often costly to produce and difficult to reuse. If the field of formal specification is to advance, we must develop sound techniques for reducing the cost of producing and reusing software specifications. The work presented in this dissertation lays out a path to producing sophisticated, automated tools for inferring large, complex code bases, tools for allowing engineers to share and reuse specifications, and specification languages for specifying information flow policies that can be written separately from program code. This dissertation introduces three main lines of research. First, I discuss a system that facilitates the authoring, sharing, and reuse of software specifications. Next, I discuss a technique which aims to reduce the cost of producing specifications by automatically inferring them. Finally, I discuss a specification language called Evidently which aims to make information flow security policies easier to write, maintain, and enforce by untangling them from the code to which they are applied.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

Software is eating the world[1]. It is responsible for entertaining us, advancing scientific research, landing planes, and in some cases, as in the case of medical devices, keeping us alive. But software is widely known to contain costly errors which can lead to losses, monetary and otherwise. As such, no question is more important than this one: How can we build systems for the future that are safe, reliable, and predicable?

In recent years advances in tools like satisfiability modulo theories (SMT) solvers have spurred significant interest in producing formally verified systems. However, a fundamental problem that plagues this line of research is that the specifications themselves are often difficult and costly to produce, often requiring specialized training that is well beyond the means of typical engineering teams [96, 12]. Even if such specialized training is available, specifications themselves are difficult to compose, share, and reuse. Furthermore, verification techniques generally require that *everything* (APIs, frameworks, etc) have a specification in order to perform sound reasoning. This presents a fundamental problem for verifying modern software systems since many such systems depend on many other libraries, frameworks, and APIs in order to perform their task. Since complete specifications of many software artifacts (such as the JDK) do not exist, any formally specified systems are likely to be unverified, and thus unsound.

The work in this dissertation is focused on addressing these problems in the context of modern software systems. That is, software systems built in common, modern languages which may rely on external software libraries and APIs to perform their function. In this dissertation I focus on the Java language due to its overwhelming popularity and rich history of research in the software

[1]Read Marc Andreessen's essay on why "Why Software Is Eating The World" from 2011, here: http://on.wsj.com/2sh0ZyG

engineering community.

In order to make the content of this dissertation more streamlined, in Chapters 2, 3 and 5 I provide the background information necessary to understand the material in that chapter. That background aside, there are several common underlying concepts that weave through all of the material in this dissertation. In the following section I provide a discussion of these common terms and concepts found throughout this dissertation.

Terms and Concepts

Modern Software Systems In this dissertation, I use the term modern software system to describe a body of software. There are several requirements for such a system to be considered "modern." First, a modern software system is written in a mature, widely-used programming language. As noted in this section, in this work I focus on the Java programming language due to its popularity and its rich history of research. Second, modern software systems are often composed of dependencies on other projects. These dependencies are often expressed declaratively, i.e., the programmer need not supply the source code of the dependency, only specify the name of the dependency and the version required. Thirdly, related to the previous point, the system should rely on a standard tool for building the final software artifact. Examples of such tools for Java include Maven, Gradle, Ant, and others.

Testing The idea of program testing is one that should be very familiar to any programmer. The testing problem, is roughly defined as follows: Given a software artifact, make an assessment of the quality of the software artifact. The meaning of quality may vary depending on the particular context. For example, in certain contexts one may be concerned with the performance of the program, the "usability" of the program, or if it meets the previously agreed on requirements.

Some types of testing aim to show that a software artifact is reasonably free of bugs, however, it is not generally possible through testing to show that a program is free of bugs. As noted by Dijkstra [27], testing can be used to demonstrate the presence of a particular instance of a bug.

Satisfiability Modulo Theories The *satisfiability* (SAT) problem is a problem which asks, given a Boolean formula if there exists an assignment of values to the formula's free Boolean variables that makes the formula true. Similarly, Satisfiability Modulo Theories (SMT) combines logical formulas with theories that are expressed in first order logic. Theories are typically constructed in such a way that they are useful for expressing data structures found in computer programs such as lists and numbers. Like the SAT problem, the essential question is if the formula provided is satisfiable, that is, is there an assignment to variables such that the formula is true? However, unlike a SAT problem, SMT problems are only decidable if the background theories are decidable as well.

Specification and Verification Whereas testing can only show the presence of particular (often anticipated or discovered) bugs, verification can go much further and ensure certain properties of a particular system. In this work, we use the term *specification* to mean a *behavioral specification*. This is to contrast it from other types of common specifications such as temporal specification, which are able to specify sequences of events and verify properties about of those sequences, e.g., that a program does not contain deadlocks. A behavioral specification is capable describing a *contract*, i.e., a set of pre- and post-conditions a piece of code must satisfy (Chapters 2 – 4), invariants that must be satisfied, or relations between runs of programs, as is the case with the specification language described in Chapter 5. The types of formal proofs possible with behavioral specifications have advantages over testing since they can subsume an infinite number of test cases, that is, they can consider all executions of a program, not just executions that occur with certain inputs. In this dissertation, we use the Java Modeling Language (JML) [56] in Chapters 2 – 4. In Chapter 5 we present a specification language of our own design called *Evidently*, which is designed to allow a user to specify the information flow properties of programs.

Summary of Content and Contributions

In this section we provide a brief summary of the content and contributions contained in each chapter of this dissertation.

Chapter 2 To answer the problem of how to creating tooling to facilitate the authoring, sharing, and reuse of specifications, in Chapter 2 we introduce a system called Spekl, which addresses this problem. We describe the motivation for such a system, describe its design, and demonstrate the problems that exist with current approaches to specification authoring, sharing, and reuse.

Chapter 3 As discussed in the introduction, for verification to be sound, one must have a specification for all of the methods that a program depends on. However, producing specifications is a costly, time-consuming activity. Furthermore, many specifications, even for popular libraries and frameworks, do not currently exist. In Chapter 3 we describe a system called Strongarm which aims to address this problem by automatically inferring the behavioral specifications of Java programs. We present our tool, Strongarm, and discuss a novel algorithm called FAR for simplifying specifications produced by forwards symbolic execution on Java programs. We provide a detailed description of FAR as well as a proof of soundness.

Chapter 4 In Chapter 4, we conduct the detailed evaluation of the specifications produced by Strongarm and the FAR algorithm. We evaluate Strongarm according to the following dimensions:

1. **Effectiveness of Inference** How many of the candidate methods we were able to infer?

2. **Efficiency of Reduction** By how much were the specifications reduced?

3. **Complexity of Inferred Specifications** Overly complex and overly simplistic specifications are not practical. What are the characteristics of the inferred specifications?

4. **Performance of Inference Procedure** How well (with respect to time) did Strongarm perform at the task of inferring specifications?

5. **Performance of FAR** How effective is FAR at reducing specification nesting, decreasing specification length, decreasing prover execution time, and decreasing proof length?

Chapter 5 Lastly, in Chapter 5 we discuss the design of a new specification language called *Evidently*which allows a user to specify information flow policies. The key innovation of *Evidently* is that it allows a user to specify the information flow policy in a way that is not scattered and tangled with the code it describes. We describe an implementation of *Evidently* which leverages runtime data tagging in the Java Virtual Machine to support enforcement of information flow security policies.

Chapter 6 This chapter concludes this dissertation.

CHAPTER 2: A LAYERED APPROACH TO SPECIFICATION AUTHORING, SHARING, AND USAGE

One of the hallmarks of modern software engineering practice is widespread code reuse. Modern engineering teams have many tools at their disposal for code reuse. For example, systems like Maven enable engineers to distribute software libraries to central libraries where they may be consumed (reused) by other engineering teams. To promote reliability and correctness, there must also be a way to compose specifications for reuse. However, specifications cannot be adapted by the use of wrappers in the same ways as code, which leads to specifications being copied and modified. This copying and modification of specifications leads to poor maintainability and technical debt.

In this chapter, we discuss the design and implementation of a system, Spekl, that solves these problems and makes compositional reuse of specifications possible in a way independent of the choice of specification languages and tools. We provide a detailed description of our system as well as provide details on our domain specific language for creating new tools, provide details on how to author new specifications, and demonstrate how Spekl facilitates compositional reuse through specification layering. The material in this chapter was largely derived from the following publication: John. L. Singleton and Gary. T. Leavens. A layered approach to specification authoring, sharing, and usage. In *Advances in Intelligent Systems and Computing*, vol. 561, 2018, pp. 164-189.

Motivation for Specification Reuse Tooling

Software libraries have become a critical component of modern software engineering practice since they promote modularity and reuse. Unfortunately, the same cannot be said for specifications. This is because software libraries are typically reused in one of two ways: directly or adaptively. In *direct* reuse, the library matches the desired functionality perfectly and is simply reused as is. In *adaptive* reuse, the developer writes code that modifies (wraps) the library's computation in a way that makes it usable within the program at hand.

However, software specifications are encumbered by several issues that impact their reuse in ways that do not impact library code (we detail these differences in Section 2). Because of these differences, existing tools are inadequate for specification authoring and distribution.

The root of the problem is that, unlike software libraries, specifications may not always be compositionally adapted by their clients (we explain several different cases in which adaptation is made difficult in Section 2). This difference often means that the specification must be copied and modified, a non-compositional form of reuse that is considered unacceptable for code. Directly modifying a copy (rather than using an adapter) negatively impacts modularity and makes it difficult to update to newer versions when they become available; hence such copy and modify adaptation is not used for code libraries. However, in the case of specifications, since an adapter is not always possible to create, the client may be forced to use the copy and modify technique, despite the maintenance problems it creates. The problem is that after modifying the copied specifications, the task of merging in ongoing work on the specification and keeping the local modifications becomes part of the technical debt of the client.

When using specifications, clients often have an existing code base that they want to check for some properties, such as safety or security. Verification of these properties must often use specifications for various libraries which are used in the code. If these library specifications are not appropriate for the verification, then they need to be adapted or modified. Furthermore, the code that clients want to check should not be modified to accommodate these specifications. Therefore, techniques that are based on code adaptation will not work.

We propose a system, Spekl,[1] aimed at solving this problem by making the software specification workflow easier to manage. Spekl addresses two key research questions:

RQ 1 How can tooling promote the reuse of pre-existing specifications?

RQ 2 How can tooling make existing specifications more useful?

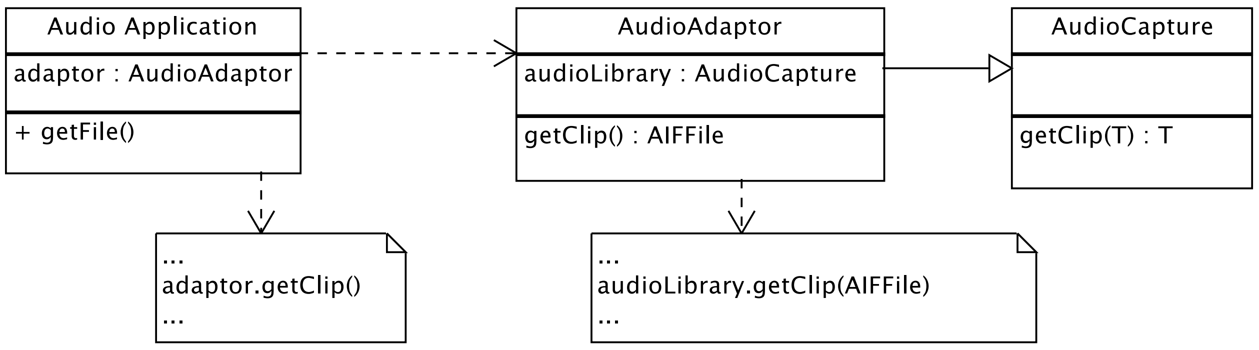
The design of Spekl is based around the hypothesis that the problem of effectively solving the specification reuse problem can be reduced to three subproblems: tool installation/usage, specification consumption, and specification authoring and distribution. Before proceeding, we emphasize that the focus of Spekl is on externalized specifications, i.e., those separate from the text of the program. Spekl does not currently provide support for internalized specifications. This is because externalizing dependencies promotes modularity and thus facilitates reuse, both of which are goals of the Spekl system.

The remainder of this chapter is organized as follows: In Section 2 we consider more deeply the motivation behind creating a tool tailored to the task of specification management. Then, in the following subsections, we explore these subproblems and explain how they relate to the design of the Spekl system. In Section 2 we review the main features of the Spekl system including verification tool installation and usage, specification consumption, and specification layering and authoring. In Section 2 we cover the details of creating new tools, provide a description of our domain specific language for writing new tools, and finally we provide details about publishing and installing tools. In Section 2 we discuss Spekl's features for consuming specifications and provide details on how the specification consumption features integrate with Spekl's tool running features. In Section 2 we discuss the specification authoring features of Spekl, including the details

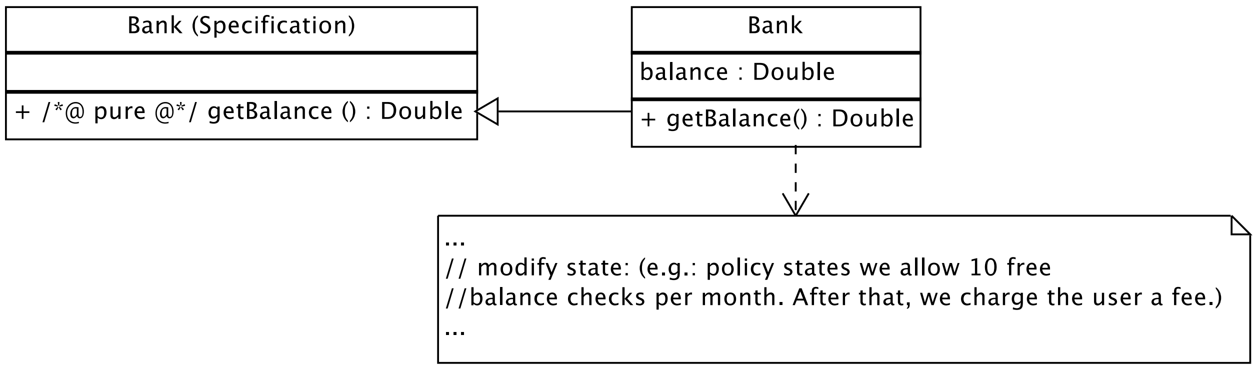
[1]The name *Spekl* comes from the abbreviation of Spekkoek, a cake originating in the Netherlands that is densely layered.

on creating new specifications as well as layering specifications. Finally, in Section 2 we provide a discussion of the work related to the Spekl system.

Problems in Specification Reuse



(a)


Bank (Specification)
+ /*@ pure @*/ getBalance () : Double
Bank
balance : Double
+ getBalance() : Double
...
// modify state: (e.g.: policy states we allow 10 free
//balance checks per month. After that, we charge the user a fee.)
...

(b)

Figure 2.1: (a) An example of an adapter that adapts the type of audio format produced by an audio capture library. (b) An example of attempting to adapt a pure method specification in an implementation that violates the purity.

To further motivate our approach, in this section we examine specific problems that impact the reuse of specifications.

Since specifications are associated with code, one might think that they could be managed using a source control system in the same way as code. However, in this section we will explain the unique aspects of working with specifications that make the direct use of such systems an incomplete solution to research questions 1 and 2.

As discussed in the introduction, the most fundamental difference between specifications and code is found in the kinds of changes one must make to each kind of artifact in order to perform adaptive reuse. For adaptive reuse of code, one can reuse existing source code without changing it by writing an adapter, e.g., a method that calls methods in an existing library to perform some task that the library does not support completely. This adaptation can be done in a purely additive manner, e.g., by adding new modules (containing new methods) that reuse existing code. For an example of this type of adaptation, see Figure 2.1a. In the UML diagram, we depict an audio application that makes use of an audio library and an adapter to produce values in a format compatible with the client program.

However, for adaptive reuse of specifications, one may need to make semantic changes to the existing specifications. Although some changes are purely adaptive, such as adding missing specifications, others will not be.

Purity

Some examples where purely adaptive reuse does not work are caused by assumptions that verification tools must make for soundness; for example, a specification language (like JML [55]) that checks methods for lack of side effects may require that such methods must be declared to be "*pure*"; methods without such a declaration are assumed to not be pure, which is the only sound assumption, because only methods declared as pure are checked for lack of side effects. (In particular a pure method may only call other pure methods.) Thus, there is no way to write an adapter if one needs to prove that a particular method is pure; a new method that is declared to be pure cannot simply call the existing (non-pure) method, as that would not verify. The only fix is to change the specification of the existing method and verify its implementation; that is, to change the existing specification in a non-additive way. For an example of this type of problem, see Figure 2.1b. In this example we show a specification for a banking application wherein a method has been marked "*pure*" in its specification. However, due to some new business requirements, a programmer must add a new behavior to the banking application that places a restriction on the number of free balance checks per month. To achieve this, purity must necessarily be violated since this method will have side effects, i.e., not all invocations of the method will produce the same results.

Specifications are Specialized to Analysis Domains

Why can't we simply create specifications that do not suffer from the problems described previously?

The task of creating a single specification to handle the needs of every conceivable user is daunting. This problem is rooted in the very nature of specification; that is, the goal is to specify a particular behavior that can be relied on. Any deviations from the base specification can cause it to become invalid and therefore unusable for verification.

To gain an intuitive understanding of the problem we will now consider an example which will not require knowledge of program verification techniques.

Consider the program and specification in Listing 2.2. This example was written in the JML [55] specification language and it specifies the behavior of a very simple function that adds two positive integers.

So what is the problem in Figure 2.2?

```
public class MaybeAdd {
   //@ requires 0 < a;
   //@ requires 0 < b;
   //@ ensures  0 < \result;
   public static int add(int a, int b){
       return a+b;
   }
   public static void main(String args[]){
       System.out.println(add(2,3));
   }
 }
```

Figure 2.2: A simple program and specification.

```
public class MaybeAdd {
   //@ requires 0 < a;
   //@ requires 0 < b;
   //@ ensures  0 < \result;
   public static int add(int a, int b){
       return a+b;
   }
   public static void main(String args[]){
       System.out.println(add(2,3));
   }
 }
```

Figure 2.3: The updated specification of Figure 2.2. Note that the preconditions have been strengthened and the bounds on a and b have been tightened.

The problem is as follows: the specification in Figure 2.2 only considers integers greater than zero. This is fine, but what if the integers are very large? An overflow condition is possible, though whether it is encountered by a runtime assertion checker depends on the test data used. A static checker however will detect this flaw and insist that the conditions in Figure 2.2 are made stronger. The code in Figure 2.4 shows the changes that are needed. Due to the changes made, the specification in Figure 2.4 does not refine the one given in Figure 2.2.

```
public class MaybeAdd {
   //@ requires 0 < a;
   //@ requires 0 < b;
   //@ ensures  0 < \result;
   public static int add(int a, int b){
       return a+b;
   }
   public static void main(String args[]){
       System.out.println(add(2,3));
   }
 }
```

Figure 2.4: The updated specification of Figure 2.2. Note that the preconditions have been strengthened and the bounds on a and b have been tightened.

Changes Invisible at the Language Level Can Cause Incompatible Specifications

In the previous sections we explained how specifications may be difficult to manage, because unlike source code, adaptation is not always possible, e.g., the method purity example in Section 2 and how writing a general specification to cover all possible use cases is difficult, e.g., the task of writing a specification flexible enough to be used effectively in runtime and static checking in the previous section. In this section, we discuss another problem that impacts specification reuse that has impacted a real-world project, OpenJML.

When using OpenJML, a user is expected to supply the specifications they want to check. To reduce programmer effort, OpenJML distributes a set of specifications for Java itself; without these specifications, any client that makes use of the Java standard libraries would be required to provide these specifications before checking their program.

Typically, between releases of Java, new library methods are added or modified. To accommodate this, the OpenJML project maintains specifications for Java 4 thru Java 7. In order to modularize the effort, these specifications are maintained separately then combined to target a particular Java release. For example, the specifications for Java 7 would be omitted for an OpenJML release targeted for Java 6.

However, in preparation for the release of Java 8, the signature of the constructor for the `Class` class was changed between Java 1.7.79 and Java 1.7.80 to include an argument to the private constructor (this was missing in all prior versions of Java).

In the case of a library, this small change would have presented no problem, i.e., private constructors are hidden, furthermore, most client code will never directly construct a `Class` object. However, in the case of a specification, the prior specification (which did not include a parameter in the constructor) is therefore invalid if a JDK higher than 1.7.79 is used. To solve the problem, a user must either manually override the specification to be compatible with their JDK or rely on a special release of OpenJML that includes the specification of the `Class` class for that particular version of Java. The main difference between the two solutions is mainly that in the first case, the burden is on the user, and in second, an additional burden is placed on the tool author.

Subverting Security Specifications

Not all specifications are strictly behavioral in nature. Some specifications, such as those designed to describe information flow [8, 30], are written specifically to prevent adaptation; if such adaptation were possible, it would not be possible to soundly enforce the security properties described by the library specifications. For example, consider an application designed to run on the Android operating system. A specification designed for use in an information flow verification tool may state that all values coming from the network should be treated as "untrusted." If an application utilizing the Android API has different security requirements and perhaps instead trusts network input, without copying and modifying the underlying specification, any calls to methods that produce values originating from the network will necessarily produce "untrusted" values. Rather than using the copy and modify approach, a more ideal solution would be to rely on tooling to remember the differences between the base specification and the enhancements made for the particular application.

Overview of Spekl's Features

Although the question of how to promote specification reuse is central to the design and motivation of Spekl, several other features such as support for installing and using tools are needed to support this goal. In the next three sections we will describe the three central features of the Spekl system: tool installation/usage, specification consumption, and finally, we will provide a description of the specification authoring and distribution features of Spekl. We provide an outline of these features in this section and provide additional details in Sections 2 - 2.

Verification Tool Installation and Usage

In order to verify a program, a user must often install appropriate verification tools. However, due to the variety of tools and technologies used to author these tools, there does not exist a general way to go about installing them. This presents a problem not only in software engineering contexts [63, 22], but also in teaching scenarios, where tools must be installed onto the computers of students. To that end, the first problem Spekl solves is the problem of distributing program verification software. Furthermore, since many verification tools such as OpenJML [15], SAW [17], and SPARTA [30] require external software packages, such as SMT solvers, in order to function properly, a single download is not sufficient to create a fully functioning verification environment.

Dependency management is a well-studied problem; on systems such as Linux, a user may choose to consume software via a variety of package management systems such as APT or YUM. While there are some cross-platform package management systems in existence, none have gained widespread usage and are therefore more likely to be detrimental to the purpose of increasing adoption of specification technologies.

Installing executable tools is only part of the problem. After a package is installed (even if a cross platform package manager were available) the user would then have to learn a series of tool-specific commands (or GUI actions) necessary to verify a piece of software. However, such specialization is not necessary. Spekl differs from a standard package management system in that Spekl provides an abstraction layer and a uniform interface to program verification tasks. Once a user configures project for Spekl, a user may run their verification tools by using the `spm` command from within the project directory.

The general command a user may use for checking their programs is the `spm check` command as shown in the following example:

```
~/my-project$ spm check
```

Executing the `spm check` command in this way causes all of the configured checks to be executed and their output displayed to the user. The uniform interface to running verification tools makes Spekl especially well-adapated for use in automated environments such as Continuous Integration.

All Spekl tool and specification packages are configured via a `package.yml` file, which is discussed in more detail in Section 2. Since the tool management aspect of Spekl is essentially that of a package manager, we provide all the usual support for tasks like dependency management, platform specific installation tasks, and packaging. Since these features are somewhat common to many systems, we omit their discussion.

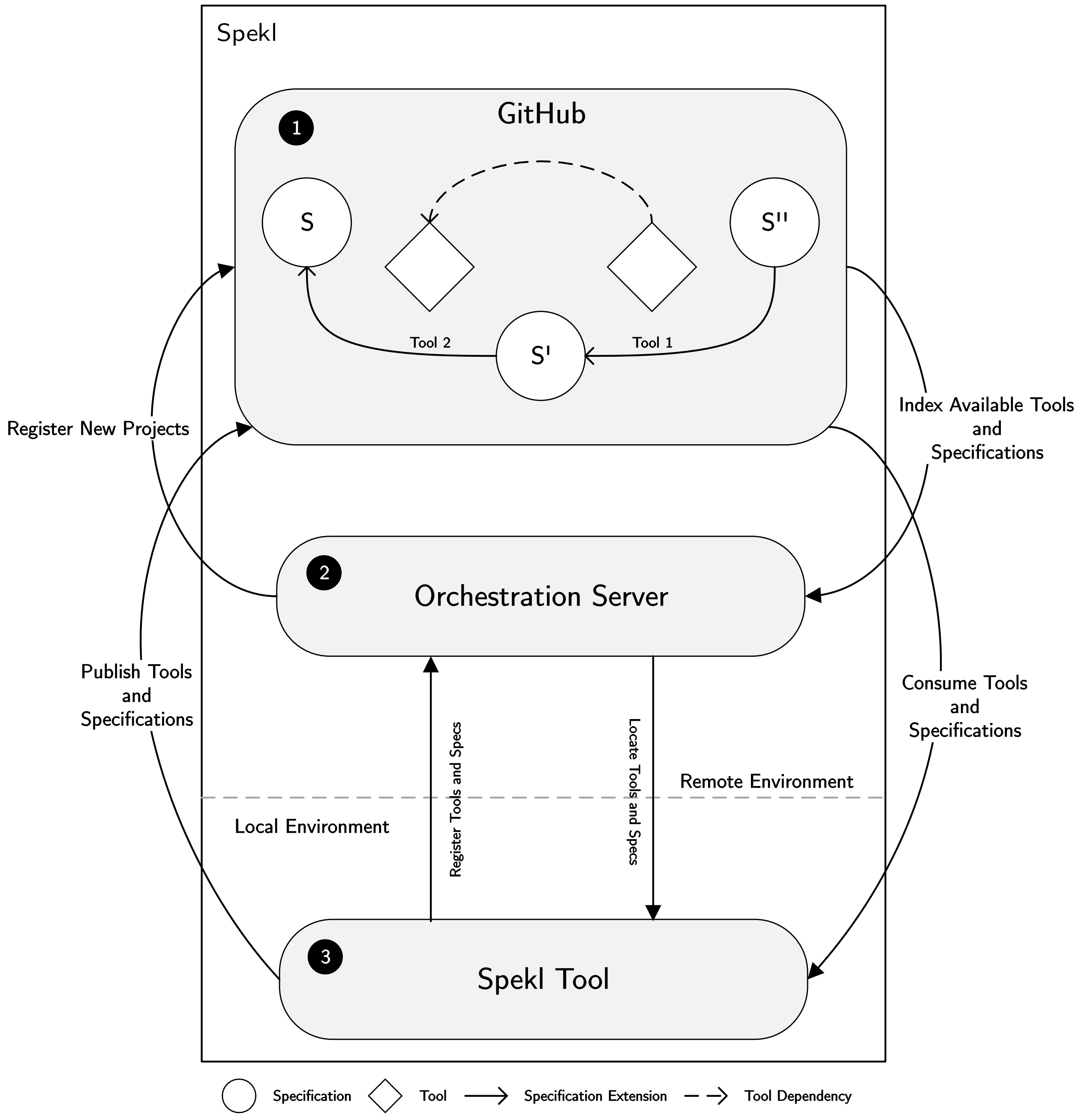


Figure 2.5: An overview of the Spekl system. Spekl relies on a vertical architecture in which a decentralized repository of specifications and tools (1) is managed by an orchestration server (2) and accessed by users via the Spekl command line tools (3).

Specification Consumption

Any non-trivial program is comprised of more than the source code written by one individual author. For example, if a user is writing an application with a graphical user interface, their application may rely on dozens of interdependent libraries and packages (e.g., Swing or SWT), all of which may require specifications to be present in order to verify the program the user is writing.

Some verification tools handle this problem by distributing the full set of specifications available for their given tool when the user performs an installation. However, this approach is problematic for a number of reasons. First, programs and libraries evolve over time. Take for example the recent flaw found in Java's implementation of the TimSort algorithm. Though the essential behavior of the sorting algorithm remained the same (it was a sorting algorithm and put items in order), the *specification* of the method did in fact change [41].

In order for the user of a program verification tool such as OpenJML to get the update to that specification they would have to download a new version of the verification tool. In the Spekl model, a user can subscribe to specific versions of named specification libraries that are automatically kept up to date by the Spekl tool. This helps users to verify their programs against the latest specifications. This is helpful not only in the case that a refined specification is discovered; large code bases (for example, the Android API) may have partial or incomplete specifications. Spekl's automatic update mechanism allows program authors to gain access to revised specifications earlier, as they become available.

Specification Layering and Authoring

Writing a specification that will be applicable to every user is a challenging (if not impossible) task. For example, suppose a user is using a taint analysis tool that classifies all input from external devices as tainted and therefore should be classified as a defect in any program that allows its input to come from external devices. Although this may be desirable in some scenarios, the authors of the specification, by being overly specific, risk making the specification unusable for other purposes. If the user of the specification would like to specify that input from some trusted external device (say, a crytographic USB key) is not tainted the user will be unable to specify this behavior without finding ways of subverting the existing specification.

Unfortunately, the way in which a user will typically perform this task is to modify the specifications directly to override the specified behavior. However, this has the disadvantage that the specifications are now orphaned from the revision history from which they are derived. Edits to the specification library will become the technical debt of the project author and any subsequent updates to the base specifications will require the author to manually merge in changes from the upstream package. Even if the user does manage to succeed in modifying the offending specification, they will have now *diverged* from the original specification; any time the user updates the tool they are using, they must remember to update the specification they have modified. To complicate matters, if the specification they have modified has also changed in the tool package, then they must manually merge their changes with the changes distributed by the tool package. The manual effort required by this process makes it brittle and error-prone.

Simply being able to access predefined software specifications is an important part of the specification adoption problem. However, an important and under-addressed aspect of the problem is what to do when the existing reference specifications do not meet the needs of the user. We refer to specifications that have been modified for use in a new context as *child* specifications. While this topic will be covered in detail in Section 2, the final feature unique to Spekl is the ability to allow users to not only modify but subsequently share specifications. A core part of our approach, as indicated in Figure 2.5, is that those child specifications are then optionally published to an external repository where they may be consumed by other users. New specifications in turn may either be freshly authored (using no existing specification library as a basis) or written by adapting existing specifications.

Since the focus of Spekl is specification authoring and reuse, in Section 2 we provide additional details the features of Spekl related to specification authoring and usage. Finally, in Sections 2 and 2 we provide details about Spekl's specification layering and extension features.

Now that we have given an overview of the Spekl system, we turn our attention to detailing the various features of Spekl. In the next we will discuss in detail the facilities that Spekl provides for verification tool management.

Verification Tool Management

The first hurdle in getting users to verify their software is installing the verification tools. This presents a problem not only in software engineering contexts [63, 22], but also in teaching scenarios, where tools must be installed onto the computers of students. To that end, the first problem Spekl solves is the problem of distributing program verification software. This section discusses how Spekl allows tool creators to author and publish tools to the Spekl repository as well how it helps users install and run these tools.

Creating New Tools

The first step to creating a new tool in Spekl is to use the `spm init` command. This command will prompt the tool author for basic metadata concerning their new tool. As a first step, Spekl will prompt the tool author for their GitHub username, under which they will publish the tool. The `init` command is executed as follows:

```
~/my-tool$ spm init tool
```

Spekl prompts the user for the tool's package name and version. If the current working directory is already a spekl-managed directory (meaning it contains a `spekl.yml` file), then a new tool is created at `.spm/<name>-<version>`. Otherwise, a new `package.yml` file is created in the current working directory. In addition to creating a new `package.yml` file, the `spm init` command also registers the tool's package name in the central Spekl repository, reserving that tool name. Spekl package names are assumed to be universally unique, thus attempting to use a preexisting package name is not allowed.

The Spekl Package Format

The `package.yml` that was created with the `spm init` command is the basis for configuring Spekl in the context of both tools and specifications. In this section we will be examining the package format as it pertains to tool authoring tasks. As a guiding example throughout this section will be referring to the example in Figure 2.6, which is the `package.yml` file for the OpenJML tool.

Package Metadata

The first section of the `package.yml` file is concerned with author metadata. In this section the tool author should indicate the name of the package, the initial version number, and the kind of package it is. The `kind` field may be one of `tool` or `spec`. As we will see later in the section on publishing packages, the `author` field of the `package.yml` file is important; an author may specify one or more authors in this section and this information will be used during the authentication process during publishing later in the package lifecycle.

```
name : openjml-esc
version : 1.7.3.20150406-3
kind : tool
description: The Extended Static Checker for OpenJML.

author:
  - name: John L. Singleton
    email: jsinglet@gmail.com

depends:
  - one-of:
    - package: z3
      version: ">= 4.3.0 && < 4.3.1"
      platform: all
    - package: why3
      version: "> 0.80"
      platform: all
  - package : openjml
    version : ">= 1.7.3 && < 1.8"
    platform : all

assets:
    - asset : MAIN
      name : openjml-dist
      url : http://jmlspecs.sourceforge.net/openjml.tar.gz
      platform: all

assumes:
   - cmd: java --version
     matches: java version "1.7.*
     message: "Requires Java, version 1.7."

install:
  - cmd: tar -zxvf MAIN
    description: Unpacking the archive...
    platform: all
  - cmd: touch openjml.properties
```

Figure 2.6: The `package.yml` file for the OpenJML-ESC package.

Expressing Package Dependencies

Many verification tools depend on the presence of other external tools. However it is not always possible for tool authors to distribute all of the binaries they may require in order to function. For example, many program verification tools require SMT solvers to be installed. Since there are many viable SMT solvers (some of which only work on certain platforms), rather than distributing all possible SMT solvers, tool authors might rely on the user to install the SMT tool of their choice. Spekl automates this process by allowing tool authors to declaratively express the external packages their tool depends on.

In Figure 2.6, the OpenJML package expresses two types of dependencies with two different nodes under the `depends` keyword. This feature supports two types of dependencies. The first kind is a required dependency, which must be satisfied prior to installation. The second kind is a `one-of` dependency. As in the example with the SMT solvers, a user may wish to install one of several different possible options. Specifying a list of dependencies in the `one-of` node of the `package.yml` file instructs Spekl to prompt the user for a choice during installation. If a given tool should only be installed on a certain operating system type, a tool author my indicate one of `windows`, `osx`, `linux`, `unix`, or `all` to indicate which operating systems require the dependency being expressed.

Package Assets

A given tool may be the amalgamation of several different software artifacts installed into a single host system. In Spekl, a tool's *assets* are the individual bits of software that make up a tool. To specify an asset a user must create a tuple of `(asset, url, platform)` under the `assets` node in `package.yml`. The `asset` attribute creates a new binding to be used later in the installation process. In the example in Listing 2.6, the `MAIN` name is bound to the asset that will be eventually downloaded from the location at `url`. This name may be used symbolically in the commands found later in the `install` section of `package.yml`.

Establishing Environmental Assumptions

In addition to the built-in dependency management system, Spekl is also able to verify environmental assumptions that must be true in order for package installation to be successful. In Figure

2.6 OpenJML assumes that the binary in the current user's path for `java` must be version 1.7. Note that environmental assumptions are different from dependencies in that they pertain to the state of the user's computer, where as dependencies pertain to the internal state of a user's set of installed packages.

To specify an environmental assumption, a user may add a node under the `assumes` node of the `package.yml` file by specifying a tuple of the form `(cmd, matches, message)`. The condition for proceeding is the logical conjunction after executing `cmd`, comparing the output with the regular expression contained in `matches`, and if the regular expression does not match the output of `cmd`, the message specified by `message` is displayed.

Specifying Installation Commands

The last section of the `package.yml` file is concerned with the actual commands necessary to install the tool onto the users' system. Unlike system-wide package installation tools like the APT package manager, Spekl installs tools and specifications on a project basis. This feature enables users to use different versions of verification tools (and different versions of their dependencies) on different projects without causing conflicts arising from installing conflicting tools (and possibly versions of tools) on the same system.[2]

To achieve this, Spekl maintains an installation database under the `.spm` directory within a Spekl project. For example, consider the directory listing shown in Figure 2.7. In this listing we see that two packages are installed: openjml and z3.

To specify the installation commands for a package the user must specify a set of tuples of the form `(cmd, platform)`. The commands will be executed as specified in the following list.

[2]The disadvantage of Spekl's technique is the use of more disk space than sharing a single installation of each tool.

```
/my-project
|
|__README
|
|__Main.java
|
|__Util.java
|
|__Tree.java
|
|__build.xml
|
|__.spm
   |
   |__openjml-1.1.1
   |  |
   |  |__package.yml
   |  |
   |  |__jmlruntime.jar
   |  |
   |  |__jmlspecs.jar
   |  |
   |  |__README
   |
   |__z3-4.3.2
      |
      |__package.yml
      |
      |__bin
      |
      |__doc
      |
      |__README
```

Figure 2.7: An example directory layout for a Spekl project. Note that packages are wholly installed under the `.spm` directory.

Dependency Resolution Prior to starting the installation of the current tool, the `depends` section of the tool will be examined and any tools that must be installed will be installed.

Database Preparation After dependencies are resolved, Spekl will create a directory hierarchy to support the installation of this package. The installation directory will be located in the `.spm/<package>-<version>` directory.

Asset Acquisition Prior to starting the installation commands specified in this section, each asset listed in the `assets` section will be downloaded to the local computer and placed in the database directory created in the previous phase.

Binding Substitution The `assets` section of the `package.yml` file also establishes bindings for later use in the installation process. In the listing in Figure 2.6, the `assets` section establishes a binding between `MAIN` and the eventual destination of the downloaded archive. This is useful since the tool author can (as shown in Figure 2.6) later use the `MAIN` binding in a command that unpacks the downloaded archive. These bindings are valid throughout the `install` section of the tool's `package.yml`.

Installation Command Execution The command set for the current platform is then extracted and executed in sequence. The sequence the commands are executed in is the sequence that they are specified in the file and in order for an installation to succeed all of the commands must succeed. The failure of any command causes the current installation to halt. Each command should be written to assume it is being executed in the installation directory for the package.

The Check Definition Language

Spekl's tool packages serve two purposes. The first is to provide a set of *checks* that the user may run on their software. The second is to provide resources (such as SMT solvers) for other checks and packages. In this section we focus on one of the most central aspects of tool authoring: defining the checks the tools are capable of running.

A Spekl tool may define any number of checks within a single package that may in turn be run by a Spekl user.

```
(ns spekl-package-manager.check
 (:require [clojure.java.io :as io]
           [clojure.tools.logging :as log]
           [spekl-package-manager.util :as util]))

(def report-file "findbugs-report.html")
(def report-file-xml "findbugs-report.xml")

(defn open-report [file]
  (log/info "Opening report...")
  (let [open-command (util/get-open-command file)]
      (apply run open-command)

      ))

(defcheck default
  (log/info  "Running findbugs... report will open after running...")
  (let [result  (run-no-output "java" "-jar" "${findbugs-3.0.1/lib/findbugs.jar}" "-textui" "-
      html" *project-files*)]
    (if (= 1 (result :exit))
      (println (result :out))
      (do
        (spit report-file (result :out))
        (open-report report-file)))))


(defcheck xml
  (log/info  "Running findbugs [XML]... report will be available at" report-file-xml "after
      running")
  (let [result  (run-no-output "java" "-jar" "${findbugs-3.0.1/lib/findbugs.jar}" "-textui" "-
      xml" *project-files*)]
    (if (= 1 (result :exit))
      (println (result :out))
      (spit report-file-xml (result :out)))))

```

Figure 2.8: An example Spekl check file from the FindBugs package. This check file defines two checks: a default check that outputs FindBugs reports in HTML format and a check named ”xml” that outputs FindBugs reports in XML format.

To specify that a package should export checks, a tool author should create a `check.clj` file in the root directory of the tool package. The check defined in Figure 2.8 shows the details of the checks available in the FindBugs [7] package. We describe the most pertinent portions of this check below.

As indicted by the `.clj` file extension, the host language of all Spekl checks is the programming language Clojure, a modern Lisp dialect that targets the Java Virtual Machine [46]. As such, tool authors have the full capacities of the Clojure language at their disposal while writing checks.

However, to make this process easier, Spekl defines a special `defcheck` form that simplifies many tasks. The features of this form are described next.

The `defcheck` Form

To declare a new check, a tool author must use the `defcheck` form as shown on lines 16 and 26 of Figure 2.8. These lines define the checks `default` and `xml`, respectively. Once defined, they can be referenced in a client's `spekl.yml` file, using the `check` configuration element. The `default` keyword has a special meaning in a Spekl check; although Spekl check files may define as many checks as they wish, all check files must at a minimum define the `default` check. The only other requirement of a check file is that they reside in the `spekl-package-manager.check` namespace, which is accomplished by the `ns` form on line 1.

The facilities that `defcheck` provides come in two forms: special variables and special forms. The special variables are:

check-configuration This variable contains the configuration information for the current check, as read from client's `spekl.yml` file. This feature is especially useful for tool authors that require additional configuration parameters to be present in order to successfully execute a verification tool.

project-files Prior to running a check, Spekl expands the entries given in the `paths` configuration element of a check (see Figure 2.9). This element may contain a list of literal paths, relative paths, or shell globs that specify collections of files. These paths are resolved on the local file system and provided to the check in the **project-files** variable. Since tools often run on collections of files, this feature allows the tool to remain decoupled from the project that is using it. In the example in Figure 2.8, the **project-files** variable is used as an argument to the FindBugs checker on lines 18 and 28.

specs This variable is similar to the **project-files** variable with a few important differences. In Spekl, specifications reside in packages that are stored in Spekl repositories. Since specifications may be found by version as well as name it is not possible to know statically where all the required specifications for a project may be located. Prior to running a check, Spekl resolves all of these specification paths and provides them to the check environment in the form of the *specs* variable (see Figure 2.9 for an example of how this variable is configured).

As mentioned in Section 2, Spekl checks are hosted in the Clojure language. This gives users access to a wide array of Clojure-specific functions as well as any code that runs on the JVM. In addition to this functionality, Spekl defines an asset resolution functionality that can be seen on lines 18 and 28, in Figure 2.8. The `run-no-output` form (and its complement, `run`), takes a variable number of string arguments that are then invoked on the host system. When one of the strings is of the form ”${pkg:asset}”, this causes Spekl to attempt to resolve the asset described between the curly braces. The token to the left of the colon is the *canonical name* of the package in which one expects the asset to reside. Since multiple versions of the same package may be installed on the users's system, Spekl automatically inspects the dependencies declared by the current tool and supplies only the package that complies with the restrictions set forth in the tool's `package.yml` file. If the left side of the colon is omitted, Spekl will attempt to resolve the resource in the currently executing tool's package. A failure to resolve any asset results in the termination of the current check.

Publishing Tools to the Spekl Repository

One of the challenges of creating a robust repository of software is simultaneously unglamorous and exceedingly hard: safely storing published packages. Rather than focus on the problem of building a computing infrastructure to support the storage of a software repository capable of handling possibly millions of users, Spekl has taken the approach of leveraging an existing service to handle the storage problem: GitHub. This comes with a number of advantages. First, using the free hosting capabilities of GitHub allow Spekl to remain both free and scalable. Second, since Spekl's packages are stored in Git repositories, it allows package authors fine-grained control over collaboration and workflow [89, 77].

Once a package has been authored, Spekl allows a user to publish their tool directly from the command line interface. To initiate the publishing process the user should execute the `publish` command of Spekl as shown in the following example.

```
˜/my-tool$ spm publish
[spm] Performing pre-publishing sanity check...
[spm] Current directory is a package directory
[spm] Creating new version: 0.0.1
[spm] Publishing changes to the repository
```

As mentioned in the previous section, all Spekl packages are publicly stored on GitHub. To that end, if an author wishes to directly access the repository (for example, to add a collaborator) they may access it immediately after publishing at the URL: https://github.com/Spekl/¡package-name¿. Note that since Spekl allows direct access to its repositories via Git, problems may arise if users maliciously edit repositories (for example, deleting them). Such actions could violate the invariants that Spekl requires to function. For the purposes of this work we assume users act in good faith and do not work to subvert Spekl.

Installing Tools from the Spekl Repository

Tool installation is the primary method by which end users of the Spekl system are expected to utilize verification tools. Any Spekl tool may be directly installed without configuring the project via the `spekl.yml` project file (covered in the next item). To do this a user may execute `spm` as follows:

```
~/my-project$ spm install openjml
```

This command begins an installation of the OpenJML tool for the current project. Note that this command takes an extra optional parameter `version`, which, if specified, tells Spekl which version of OpenJML to install.

While a tool may be directly installed as shown above, most users will install tools via the `spekl.yml` file. This will be covered in more detail in the next section, but a tool may be flagged for installation by being listed as a required *check* for a project.

Since Spekl is a centralized repository, the `spm` command takes advantage of this fact and provides users with a way to locate packages to install. The command that shows a user all available packages can be see in the following listing:

```
~/my-project$ spm list tools
```

This command will print a listing of the newest versions of tools available on Spekl along with the version numbers and descriptions of each tool.

Consuming Specifications

Once users are able to install verification tools they will need to combine them with specifications. The following sections are concerned with the problem of consuming specifications. We will begin with a description of how to create new Spekl projects.

```
  ~/my-project$ spm init
```

In Spekl, the normal use case for end users of Spekl is to consume packages (tools and specifications) for use in their own projects. Similar to tool creation in the previous section, the way a user may indicate an existing project should be used with Spekl is to use the `init` command of `spm` as shown in the following example.

This command prompts the user for some basic metadata about their project and creates a new `spekl.yml` file in the directory that the `spm` command was executed in.

The Spekl Specification Project Format

As discussed in the previous section, the basis for consuming specifications and tools happens at the project level and is configured via the `spekl.yml` file. This section will look at the `spekl.yml` file in detail.

Package Metadata

Unlike the `package.yml` metadata, the metadata section of the `spekl.yml` file does not contain information that will be made publicly available. Often a project will not be published (e.g., if it is internal to a company). As such the metadata section may be customized with whatever

```
name : My Project
project-id : my.test.project
version : 0.1

checks :
  - name : openjml-rac
    language : java
    paths : [src/**.java]

    tool:
      name : openjml-rac
      version : 1.1.12
      pre_check :
      post_check:

    specs:
      - java-core: 1.7
```

Figure 2.9: An example `spekl.yml` project file.

information is useful for describing a project.

Specification and Tool Configuration

The bulk of work editing a `spekl.yml` file is concerned with adding checks and specifications to the current project. This is done by manually editing the `checks` section of the `spekl.yml` file and will be the topic of this section.

One key feature of Spekl is its support for multiple checkers within a single project. This is especially useful for large, complex code-bases where one verification tool may not be enough to handle all verification concerns. For example, certain critical portions of the code base may need to be checked with static checking, which is more precise, but more demanding in terms of specifications, whereas it may be sufficient to check the remainder of the code base with runtime assertion checking.

To configure a checker, a user adds a node below the `checks` node of the `spekl.yml` file. In

Figure 2.9 we can see the OpenJML Runtime Assertion Checker configured for a project. A few items in this configuration are of note.

name The name attribute of a check is used to identify the check to the user at runtime and any suitable string may be used.

language Certain checkers may take advantage of knowing the input language of a project ahead of time. This field is optional and should only be specified if a tool needs it.

paths The *paths* element of a check is a critical component of its configuration and is the method by which the verification tool will find source files to check. The paths element is a comma-separated list of shell globs that should contain all of the files expected to be passed to the backend verification tool. In the example, the double star pattern (`**`) means to match any directory within `src/` before matching the file pattern.

The next section of the check configuration pertains to selecting a tool and version to use. When `spm check` is run it will consult this section first to ensure that all dependencies are fulfilled before running the check. The parameters of this section are as follows:

name The name of the tool to use as reported by the `spm` list command.

version The version of the tool to require. In addition to numerical version numbers, this attribute also supports symbolic version number strings. For example to specify that version greater than 4.3 is required, a user can supply the version string ”¿ 4.3”. Both conjunction and disjunction are supported and version strings such as ”¿ 4.3 && ¡ 5.0” may be supplied. The operators supported are the binary comparison operators: ¿, ¡, ==, ¡=, and ¿=.

pre_check/post_check Though not normally needed, the pre_check and post_check configuration parameters are hooks to allow a user to run custom actions before and after a check. For this parameter, a user may specify a shell command to execute. No validation on this input is performed but a failing pre_check (a command returning a non-zero exit code) will cause the check's execution to halt.

Running Verification Tools

Once the `spekl.yml` file is configured, a user may run their verification tools by using the `spm` command from within the directory where the `spekl.yml` file is located. In this section we provide details on this process.

The general command a user may use for checking their programs is the `spm check` command as shown in the following example:

```
~/my-project$ spm check
```

Executing the `spm check` command in this way causes all of the configured checks in the `spekl.yml` file to be executed in sequence. If the user has multiple checks configured and only wishes to run one of the checks, they may refine the behavior of the `spm check` command by naming the check to run:

```
~/my-project$ spm check openjml-rac
```

As shown in Figure 2.9, the configured check name is `openjml-rac`. Specifying `openjml-rac` as an argument to `spm check` indicates that the `spm` tool should only run the that specific check and no other.

Specification Authoring Features

Similar to the facilities for tool authoring, Spekl also provides support for specification authoring. Specifically, Spekl provides support for two modes of specification authoring: *specification creation* and *specification extension*.

In *specification creation*, a specification author creates a new specification for a body of code from scratch. The code being specified need not be the work of the specification author.

In *specification extension*, as we shall see in the following sections, a specification author creates a new specification based on an existing specification within the Spekl repository. The reason an author may do this is two-fold. First, the author may need to change the meaning of an existing specification to fit their needs. The second case is that the author may need to *add missing specifications* to the specification library to cover portions of the codebase not handled by the original specification author.

Creating New Specifications

To create a new specification from scratch, a specification author uses the `spm init` command as shown previously. The command is invoked as follows:

```
~/my-project$ spm init spec
```

After prompting the user for some basic metadata about the specification, the `spm` tool will create a `package.yml` file either in the current directory or in the `.spm` directory, depending on how the command was invoked before. Much of the `package.yml` file is identical to the one used for tools. The main difference is that for specifications the sections for assets, installation steps, and dependencies do not apply. Rather, the specifications must be wholly self-contained within the package itself. Similar to tools, completed specifications that are either extended or created from scratch may be published to the Spekl repository with the `spm publish` command.

Layering Specifications

In Section 2 we presented several examples where adapting a specification was not possible. In this case, rather than *modifying* a specification, what is needed is the ability to *extend* the specification in such a way that it can override the specification from which it is derived. This is a core idea behind how Spekl allows specification writers to use and extend specifications. In this section we give an example of this usage.

To begin, a user wishing to extend a specification should execute the `extend` command of `spm` with the name of the package specification they wish to extend as an argument. In our example we will extend `jml-core-java-7` which is the set of specifications in JML for Java 7.

```
~/my-project$ spm extend jml-core-java-7
```

Executing this command creates a new specification project in the directory from which it was invoked. Let us take a look at the `package.yml` file that was created.

```
name : my-jml-java-7
version: 0.0.1
kind : tool
extends: jml-core-java-7
```

Listing 2.1: An excerpt from the `package.yml` file generated by the `spm extend` command

The file is shown in Listing 2.1. As can be seen, this command adds the `extends` keyword to the `package.yml` file. The effect of this keyword has is to introduce a link between the revision history of the package it extends and the history of the new package. This keyword instructs Spekl that it should attempt to *merge* the revision histories of the parent and derived specifications before checking a specification. Any additions or modifications that have been made to the parent specification will automatically be made available in the derived specification. This *derived specification*, which may contain zero or more modifications, may then be published to the Spekl repository if desired and consumed by other authors who may find the modifications of the specification useful.

How Spekl Manages Hierarchies

One idea that Spekl introduces is the concept of *specification hierarchies*. In this section we describe specification hierarchies in detail and provide details on how specifications propagate throughout the hierarchy.

To begin, we consider a revision history of some specification S, $\mathscr{H}$, where $\mathscr{H}$ is a temporally ordered list of *revision strings* that specify the differences between two adjacent revisions in a history. An individual element of $\mathscr{H}$ is some element δ_n which is computed by taking the difference between the $n-1^{th}$ revision of S and the n^{th} revision of S. The initial difference in $\mathscr{H}$, δ_0, is defined as the empty revision string. Given an initial specification S and a revision to S, S', we write the difference between S and S' as $S - S'$.

A *specification hierarchy* is defined as the semilattice of pairs $(S, \mathscr{H})$, ordered by the refines relation, $\sqsupseteq$, that is:

$$(S', \mathscr{H}') \sqsupseteq (S, \mathscr{H}) \iff S' <: S \land (\exists \delta \in \mathscr{H}' \ :: \ \delta \in \mathscr{H}) \tag{2.1}$$

The meaning of the $S' <: S$ is that S' is declared to extend S. In Equation (2.3) and below we refer to a specification hierarchy by the more compact notation, $\overrightarrow{SH}$, which is expanded more explicitly in (2.2).

$$\overrightarrow{SH} = \{(S_\bot, \mathscr{H}_\bot), \ldots, (S_\top, \mathscr{H}_\top)\} \tag{2.2}$$

$$\textbf{extends}(S, \overrightarrow{SH}) = \left\{ \begin{array}{ll} \textit{nil} & \text{if } S = S_\top \\ (s,h) \in \overrightarrow{SH} \ : \ S <: s & \text{otherwise.} \end{array} \right\} \tag{2.3}$$

In Equation (2.2) the symbols $\top$ and $\bot$ refer to the specification at the top of the hierarchy (extended by every specification in the hierarchy) and the specification at the bottom of the hierarchy. Both $\top$ and $\bot$ must be uniquely defined and exist.

Note that in the case of the $\bot$ specification this does not imply that no specification extends it, but only that for a particular context there is no specification that extends it.

Specification hierarchies are useful in that they allow new specifications to be based off of existing specifications, thus benefiting from the pre-existing work in creating the base specification. We call specifications based off of existing specifications *child specifications*. Conversely, the direct specification on which a child specification is based is called the *parent specification*. We refer to the collection of all parent specifications (that is, the ancestor chain) as the *upstream specifications*.

What is unique about Spekl's specification hierarchies is that they allow a given specification to be modified while maintaining a historical connection to the revision history of any upstream specification that it extends. Any changes made to the upstream specification are automatically propagated down the chain. We refer to the current version of a given specification with all upstream changes applied as the *effective specification* of S.

Suppose now that a specification S' is created based on the specification S. The user extending the specification makes changes to S'. Meanwhile, the original author of S makes changes to it as well. These changes may include additions, deletions, or refinements of specifications. To complicate matters further, the upstream specification itself may in turn be based off another specification. How can we provide an appropriate effective specification?

To answer this question, Spekl defines two orthogonal concepts: specification refreshes and specification merges.

A *specification refresh* is an operation invoked by Spekl during specification checking that inspects the current specification, determines if there have been changes in the specification extension chain and applies those changes if needed in order to arrive at a new effective specification.

We provide a definition of the refresh operation in Equation (2.4).

$$\textbf{refresh}((S, \mathscr{H}), \overrightarrow{SH}) = \left\{ \begin{array}{ll} \mathscr{H}, & \text{if } \textbf{extends}(S, \overrightarrow{SH}) = nil \\ \textbf{let } (S', \mathscr{H}') = \textbf{extends}(S, \overrightarrow{SH}) & \text{otherwise.} \\ \quad \textbf{in refresh}((S', \mathscr{H} \lhd \mathscr{H}'), \overrightarrow{SH}) & \end{array} \right\} \tag{2.4}$$

The second concept, *specification merges*, is introduced by the presence of the operator $\lhd$ in Equation (2.4) — the *merge* operator. The merge operation in Spekl ($\lhd$) is in fact the same three-way merge operation used by the version control system Git, which is based on the finding the longest common subsequence between two revisions of a file. It is different than the more simplistic two-way merge in that it takes into account the information about common ancestors in a revision history [65]. Improving the usefulness of the merge operation is a task relegated to future work for Spekl.

However, in order to understand how this merge operation impacts specifications, let us consider the possibilities of merging two arbitrary specifications and histories $(S, \mathscr{H})$ and $(S', \mathscr{H}')$:

Case 1 $(\forall \delta \in \mathscr{H}' :: \delta \notin \mathscr{H})$ **Action**: No specification merge is possible since S' was not derived from S. All of the remaining cases assume that case 1 does not hold.

Case 2 $(\mathscr{H} \subseteq \mathscr{H}')$ **Action**: No action is needed since all of the history for S is already subsumed by the history of S'.

Case 3 $(\exists \delta \in \mathscr{H} :: \delta \notin \mathscr{H}')$ **Action**: There has been at least one change to S that has not been applied to S'. These changes will be merged into S' by using the three-way merge algorithm described earlier. Note that this condition assumes that $(S', \mathscr{H}')$ refines $(S, \mathscr{H})$, i.e., the ordering relation $\sqsupseteq$ from Equation (2.1) holds.

In Spekl, specification refreshes are invoked in two ways. The first way is automatically during the `spm check` command. During the execution of `spm check`, Spekl automatically collects the required specifications for a check and checks the remote repositories of the specifications to determine if they need to be refreshed. Additionally, if during specification development, an author wishes to synchronize their work with the upstream specifications, they may invoke the `spm refresh` command to manually trigger a refresh. The effective specification then becomes part of the permanent revision history of the child specification.

Related Work

The main related work on specification management (research questions 1 and 2) is that of source code control systems, such as Git. The reasons why such source code control systems are inadequate for using and extending specifications were discussed in Sections 2 and 2.

Although Spekl does aim to abstract the process of installing and using verification tools, it does not aim to be a general purpose package management system. Nevertheless, Spekl implements many of the functions that are required by a software package management system and is therefore comparable to previous work in that domain. Though Spekl concerns itself with the installation of software artifacts like package management systems such as YUM [102], APT [101], and Haskell's Cabal [21, 49], the installation of artifacts is only one of Spekl's features whereas they are the main focus of typical package management systems.

Recent work by Tucker et al. [91] demonstrated the usefulness of integrating SMT solvers into the problem of solving dependency-related problems in software package repositories. This work was later extended by Ignatiev et al. [48] who demonstrated an improvement over the OPIUM system in the case of SMT solver timeouts. Spekl's current dependency resolution strategy does not use any of these sophisticated techniques but rather uses an optimistic approach in which a package installation proceeds until it cannot. For example, if a package expresses that it requires a version of tool T greater than 1.0 Spekl will install the newest version of T satisfying that requirement. In theory, it could be the case that installing an older version of T (greater than version 1.0) could satisfy some other dependency at a later stage in the dependency resolution stage. This sort of scenario could be detected by an SMT solver-based approach and is more comprehensive. We intend to implement this strategy in a later revision of Spekl.

Another issue that impacts Spekl is the problem of broken components in evolving software repositories. This issue is explored in depth by Vouillon [92] who investigates strategies for fixing ”broken sets” in evolving software repositories. From the perspective of Spekl, the techniques explored in this work could be applied to Spekl's specification libraries. In the case of Spekl, the problem is not as straightforward as a constraint solving problem pertaining to versions of software packages, but rather tied to ensuring that software artifacts satisfy their specifications as the repository evolves. This idea is indeed promising, but has been relegated to future work for Spekl.

CHAPTER 3: TRANSMUTING PREDICATE TRANSFORMER RESULTS TO INFER CONCISE SPECIFICATIONS

In the previous chapter we described a system for sharing, authoring, and reusing specifications. However, a fundamental problem the specification community faces is the problem of obtaining the specifications in the first place. This is a problem because for deductive verification to be sound, one needs specifications for all of the library methods (APIs) that the program uses. Since modern software systems that exhibit a high degree of code reuse often depend on many external libraries, it is critical to have specifications for these libraries. However, manually writing pre- and postconditions to document the behavior of a large library is a time-consuming task; what is needed is a way to automatically infer them.

Conventional wisdom is that, if one has preconditions, then one can use the strongest postcondition predicate transformer (SP) to infer postconditions. However, we have performed a study using 2,300 methods in 7 popular Java libraries, and found, for the first time, that SP yields postconditions that are exponentially large, which makes them difficult to use, either by humans or by tools.

In this chapter we describe a solution to this problem using a novel algorithm and tool for inferring method postconditions, using the SP, and transmuting the inferred postconditions to make them more concise.

Defining the Specification Inference Problem

Understanding a complex application programming interface (API) can be a daunting maintenance task. Source code is not always available to help this task, and even when it is available it is not always helpful. Because of complex interdependencies in code, the understanding process may be non-modular. However, when pre- and postcondition specifications are available, then an API can be understood in a modular way. Unfortunately, pre- and postcondition specifications are not often available, even for widely-used APIs. The main reason seems to be that the cost of writing such specifications is similar to the cost of writing the code itself [58].

There is a rich body of work on specification mining and inference [39, 37, 71, 23, 97, 1, 81, 83, 79]. A major category of these approaches analyze call sites of an API method to collect the set of predicates at each of these call sites and then use mining techniques such as frequent items mining to infer preconditions [72, 78]. Another body of work has focused on analyzing call sites to mine temporal patterns over API method calls, e.g. [43, 73, 93, 61, 95, 66]. These works have not focused on inferring postconditions. The second kind of technique uses static or dynamic analysis on the code of the API method to infer specifications; e.g. Cousot *et al.* uses abstract interpretation [19] to infer postconditions, and Buse *et al.* uses symbolic execution [16] to infer conditions leading to exceptions. The third kind of techniques uses *dynamic* approaches to mining specifications [2, 20, 24, 31, 38, 60, 62, 76, 94, 100]. Some of these works infer temporal patterns over API method calls [100, 60, 38, 62], object-usage specifications [76], and others strengthen existing specifications [94]. Some work [31] can infer postconditions, but depends on the presence of an adequate test suite [45, 44, 74].

The main contribution of this work is a postcondition inference technique to lower the cost of producing specifications. Our technique for method postcondition inference is based on the strongest postcondition predicate transformer: starting from a precondition (e.g., `true`) this uses the body of a method to produce a logical formula that can be converted into a specification as shown in Figure 3.1c. A key aspect of our work is a novel algorithm for simplifying the often complex specifications that result from the strongest postcondition transformer (SP), which is described in Section 3. Note that the specifications inferred by our technique do not necessarily capture the intent of the programmer; rather, these inferred specifications reflect the code as it is written.

```
//@ requires true;
public int cmp(int a, int b){
    int c = a;
    if (c < b) {
        return -1;
    } else {
        if (c > b) {
            return 1;
        }
        return 0;
    }
 }
```

(a) The cmp function.

```
   public normal_behavior
     requires true;
     {|
       requires (c < b);
       ensures true;
       ensures \result == -1;
       ensures c == a;
     also
     {|
       requires !(c < b);
       requires (c > b);
       ensures true;
       ensures \result == 1;
       ensures c == a;
     also
       requires !(c < b);
       requires !(c > b);
       ensures true;
       ensures \result == 0;
       ensures c == a;
    |}
   |}
```

(b) The specification before our technique is applied.

```
public normal_behavior
  requires !(a < b);
  {|
    requires (a > b);
    ensures \result == 1;
  also
    requires !(a > b);
    ensures \result == 0;
  |}
  also
    requires (a < b);
    ensures \result == -1;
```

(c) The specification after our technique is applied.

Figure 3.1: Demonstrating FAR on the cmp procedure.

Human review of the inferred, more concise specifications can then identify errors in implementation.

An illustration of our technique that shows the inferred specification for the method in Figure 3.1a is shown in Figure 3.1c.

The key contributions of this work include:

1. A novel approach for statically deriving postconditions in the form of contracts written in a formal specification language such as the Java Modeling Language (JML).
2. The implementation of a tool, **Strongarm**, that implements our technique as an extension to the OpenJML [55, 15] program verification tool. Our tool will be part of a future OpenJML release.
3. A series of techniques for producing concise specifications and a novel, graph-based algorithm for producing shorter specifications.
4. An experimental evaluation of Strongarm on seven popular Java libraries that demonstrates our method's effectiveness at producing automatically inferred specifications.

In the next section we will examine the technical ideas that underpin our approach. In Section 3 we introduce the FAR algorithm, which can make inferred specifications more concise. In Section 4 we present an evaluation of our approach. Finally, in Section 4 we discuss the work related to our proposal.

Problems with Specifications Inferred with SP

Much research has been focused on the mining of specifications [39, 37, 71, 23, 97, 1, 81, 83, 79] and preconditions [72, 78]. Conventional wisdom is that, if one has preconditions, then one can use the strongest postcondition predicate transformer (SP) to infer postconditions. However, we have performed a study using 2,300 methods in 7 popular Java libraries, and found, for the first time, that SP yields postconditions that are exponentially large, which makes them difficult to use, either by humans or by tools.

The standard technique for statically computing the postcondition for a piece of code was first formalized by Dijkstra in the form of *predicate transformers* [28][1], essentially functions that can take a piece of code and either a pre- or post-state and produce a logical proposition that describes the effect of the code. Several researchers have since used these predicate transformers as the basis of building verification tools such as ESC/Java [34], Boogie [59], OpenJML [15], and many others.

We now present an example that details some of the problems in producing specifications by applying predicate transformers.

Consider Figure 3.1b. That Figure shows the contract produced after running symbolic execution on the `cmp` function.

There are problems to note. The first problem to note is the propagation of the precondition **`true`**. This formula represents the precondition of the method[2]. Because of the predicate transformer semantics, when a branch is encountered, the precondition is propagated to both clauses as can

[1]An equivalent and widely-used formulation of the Dijkstra's predicate transformers is the proof logic developed by Floyd and Hoare [35, 47]. The main difference between these approaches is that Dijkstra's formulation provides a precise method for symbolically evaluating a given program both *forwards* and *backwards*.

[2]Note that the precondition of `true` is chosen for brevity. In practice a more descriptive precondition would typically be written.

been seen in Equation 3.4. In that rule, the precondition P is propagated to each clause of the resulting disjunction. This in effect propagates the precondition downwards in the contract. Since the current set of conditions becomes part of the next level of nesting, the current set of propositions are propagated (as the new precondition) downwards. This process is repeated recursively for nested conditionals, effectively creating a "snowballing" effect with respect to the size of the final formula. This effect is then especially pronounced if there is a lot of nesting in the code being analyzed and would negatively impact the conciseness of the contract.

Another problem that can be seen in Figure 3.1b is a problem shared with many specification languages: a contract cannot contain references to local variables. According to the semantics of Java, local variables are not visible to clients of the method. However, strictly following the predicate transformer semantics for the `cmp` procedure produces a reference to `c`, which is a local variable.

Even with a small example, some of the problems in applying predicate transformers to the task of generating specifications are immediately apparent. Other problems, which Strongarm solves, only become apparent from experience with larger studies such as the ones presented in this paper. We have identified the following categories of problems encountered when computing specifications with sp:

Unfeasible Specification Cases Given a set of preconditions, a specification case may be unfeasible if the precondition in a prior (parent) specification case makes the precondition for a nested specification case impossible to logically satisfy.

Duplicate / Redundant Preconditions If a given piece of code has branching, the semantics of *sp* require that the precondition is propagated down all of the branches of the conditional statement. This effect was previously shown in Figure 3.1c. To correctness with respect to the other *sp* forms this detail is necessary for the computation of the strongest postcondition.

$$\texttt{sp SKIP}\ P = P \tag{3.1}$$
$$\texttt{sp}\ (V := E)\ P = \exists v.(V = E[v/V]) \wedge P[v/V] \tag{3.2}$$
$$\texttt{sp}\ (S_1; S_2)\ P = \texttt{sp}\ S_2(\texttt{sp}\ S_1 P) \tag{3.3}$$
$$\begin{aligned}\texttt{sp}\ (\texttt{IF}\ B\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2)\ P = (\texttt{sp}\ S_1(P \wedge B)) \vee \\ (\texttt{sp}\ S_2(P \wedge \neg B))\end{aligned} \tag{3.4}$$
$$\begin{aligned}\texttt{sp}\ (\texttt{WHILE}\ B\ \texttt{DO}\ S)\ P = (\texttt{sp}\ (\texttt{WHILE}\ B\ \texttt{DO}\ S)(\texttt{sp}\ S\ (P \wedge B))) \\ \vee (P \wedge \neg B)\end{aligned} \tag{3.5}$$

Figure 3.2: Predicate transformer rules used by Strongarm's symbolic execution engine.

However, once the strongest postcondition is computed these preconditions can often be removed from the resulting clauses.

Tautologies Some specification inferences result in vacuous clauses.

Frame Axioms One issue outside the scope of predicate transformers (but handled by many specification languages) is the problem of Frame Axioms, which are used to precisely describe how the program's state is modified by a procedure.

Purity Another feature not addressed by predicate transformers is the issue if Method Purity, which is a specification language feature used to allow specification authors to use functions that have no side effects within specifications. This is an important feature of specification languages since it allows specifiers added expressiveness.

In the next section we will discuss our approach to solving the problems encountered when inferring specifications with *sp*.

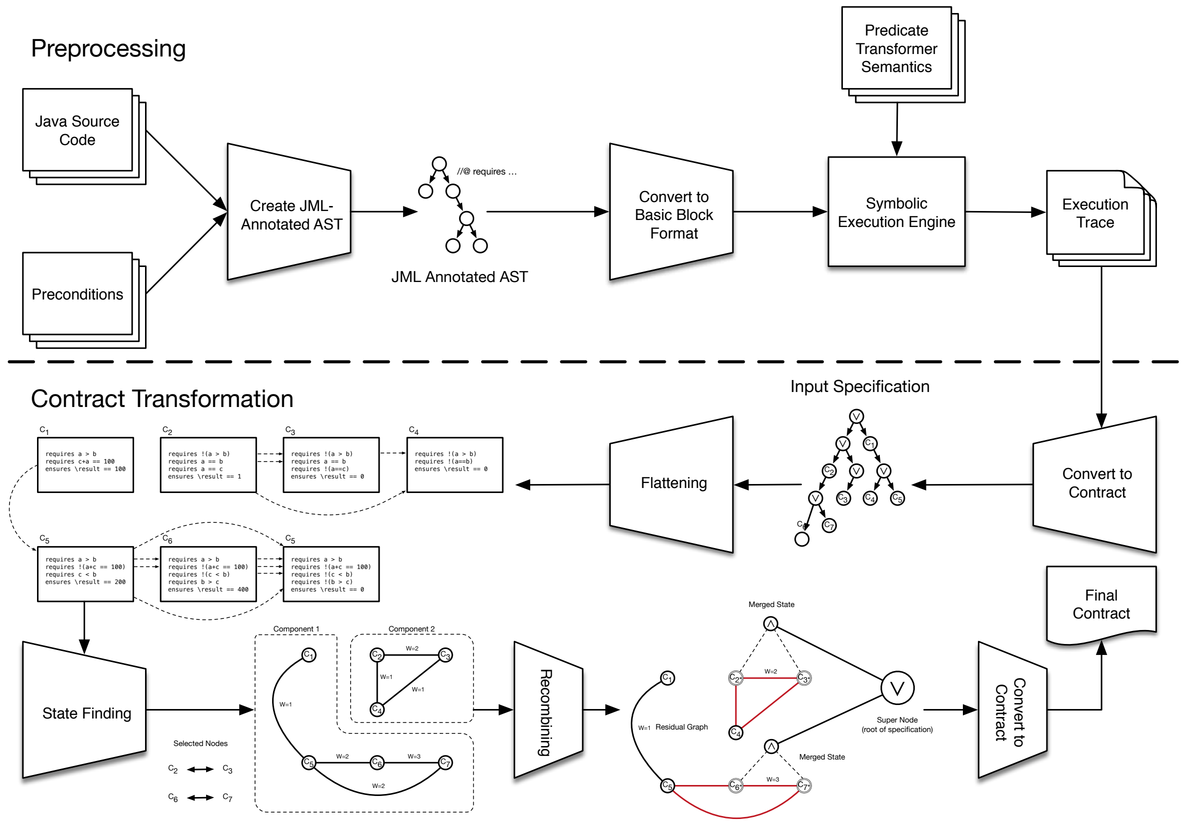


Figure 3.3: The Strongarm specification inference architecture. Strongarm's design is broken into two phases: Preprocessing, which handles the initial program transformation to Optimal Passive Form as well as the symbolic execution of the transformed program. The execution trace from the first phase is then given as input to the post-processing stages which are responsible for further processing the inferred specification. In the contract transformation phase we omit the details of the base analysis types (Section 11) and focus on the four steps of FAR: Input specification, flattening, state finding, and recombining.

Our Approach

In this Section we will be discussing the technical details of Strongarm. It is built as an extension to the OpenJML verification tool; in addition to inferring postconditions, Strongarm's design allows enables it to take advantage of pre-existing specifications and supports most JML specification language features.

First, we need to solve the problem with existential quantifier in SP. Consider for example the assignment rule in Equation 3.2 in Figure 3.2. The meaning of this equation is that it calculates the strongest postcondition (sp) of the expression $V := E$ when it is executed in a state where P is said to hold. Note that Equation 3.2 uses an existential quantifier. This is problematic because it requires the use of a constraint solver. We avoid this problem by using a type of program representation called the Optimal Passive Form [42, 59, 10]. In addition to simplifying the control flow graph of a program, the key feature of this form is that it uses a single static assignment (SSA) format, which, because every assignment to a variable results in a new variable, eliminates the need for existential quantifiers.

Our approach is based on the following three key technical insights:

Predicate Transformers Guide Symbolic Execution. In this paper we have been discussing Dijkstra's predicate transformers as a foundation for computing postconditions. An equivalent and widely-used formulation of the Dijkstra's predicate transformers is the proof logic developed by Hoare and Floyd [47, 36]. The main difference between these two approaches is that Dijkstra's formulation provides a precise method for symbolically evaluating a given program. In the case of *weakest preconditions*, the approach favored by many verification tools [10], computing the weakest precondition is equivalent to backwards symbolic execution. In this case of *strongest postconditions* (what we are after), it is equivalent to forwards symbolic execution [40]. In our approach, rather than relying on traditional symbolic execution approaches, which are subject to problems with state space explosion and issues with non-linear inequalities, we instead base our approach on predicate transformers. The rules we used in our work are detailed in Figure 3.2. In addition to avoiding state space explosion, our approach has been formulated to efficiently leverage the Optimal Passive Form already existent in OpenJML as well as other verification tools.

Defer Optimization Until After Symbolic Execution. In our system, additional optimization to the final contracts produced by Strongarm can be influenced in one of two ways. As seen in Figure 3.3, the symbolic execution engine takes in a set of predicate transformer semantics; the resulting execution trace is handed to a specification simplification pipeline that employs a variety of modular techniques to produce readable contracts. Thus, in order to influence the output of the system one can either enhance the semantics that drive the symbolic execution engine or add additional transformations to post-execution processing pipeline. In our design, we always opt to add this complexity to the post-processing pipeline.

Formulas Computed with sp are Implicitly Parallel. A key technical insight of this work is our observation that formulas computed via the *sp* predicate transformer produce formulas that can be easily made more concise. The details of this technique are discussed in Section 3.

In the next section we will discuss one of the key contributions of this work, an algorithm that addresses the problems introduced in Section 3.

The FAR Algorithm

The main problem with formulas computed by *sp* is that when the results are converted into specifications, they can become very large, deeply nested, and repetitive. At each branch point in the program the branch's precondition is fed back into *sp* (see Equation 3.4) as an argument, which causes the number of specification cases in the resulting specification to grow exponentially. In this section we describe the Flattening and Recombination (FAR) algorithm that produces an equivalent specification that is shorter and more practical than the input specification.

In this section we discuss the FAR algorithm and soundness of the results produced by FAR and develop a theorem about its soundness. We say that FAR is *sound* if the specification that results from running FAR is equivalent to its argument specification. By *equivalent* we mean that the set of programs satisfying the argument specification is the same as the set of programs satisfying the resulting specification.

The outline of this section is as follows. First, in Section 3, we discuss the connection between JML specifications and first order Boolean formulas and describe our meaning of the specifications that FAR operates on. As we shall discuss, the soundness of FAR depends on the soundness of $\sim$, and in Section 11 we will discuss the soundness property of $\sim$ used in FAR. Lastly, in Section 11 we discuss the soundness theorem of FAR.

Abstract Specifications

In discussing the soundness of FAR, it is useful to abstract the specifications FAR operates on from particular specification languages. In this section we discuss the formal details of the kinds of specifications FAR operates on and produces independent of a particular specification language. The specifications themselves are to be thought of as method specifications. The only interesting properties we assume about specifications are that one can extract cases (such as those produced by SP) from a specification and that each such case has a set of preconditions (conceptually a conjunction) and a set of other clauses (also conjoined). Thus each specification is modeled as a non-empty list of cases, separated by a special operator $\vee$. Within a case, each clause is modeled as an "atomic formula," which can be thought of as an assertion whose structure is not further examined; FAR treats atomic formulas as variables. Specifications allow preconditions to distribute over cases, but in specification normal form (SNF), which is the format of the output of *sp* and the input of FAR, no such distribution of preconditions is allowed; thus in SNF each case is a non-empty list of preconditions and other clauses, each separated by the operator $\wedge$. However, in specifications in general $\wedge$ can be used to distribute preconditions over other specifications.

The semantics of specifications is most easily understood for the simple case of SNF, where the intuition is that for a specification of the form $a \vee b \vee c$, if the preconditions of each of the cases a, b, and c are never true in the same pre-state, then an implementation that transforms a pre-state s into a post-state s' is correct if the pair of states (s, s') satisfies the non-preconditions of the case whose precondition was true in the pre-state s. This semantics the same as JML's `also` operator when the preconditions are disjoint, and the outputs of SP and FAR always have disjoint preconditions. A specification of the form $(\texttt{pre} \bigwedge p) \wedge S$, is an abbreviation for distributing the precondition p into each case of S, and thus simply adds the precondition p to each specification case of S.

Definition 1 (A Specification Language)**.** A *specification language* is a language modeled by a set of expressions, *Exp*, a set of type expressions, *TypeExp*, that includes *Boolean*, a set of type environments, *TypeEnv*, a typing judgment, $\vdash$, a set of states, *State*, and, for each type environment $\Gamma \in$ *TypeEnv* a meaning function for expressions, $\mathcal{E}_\Gamma$. Type environments $\Gamma \in$ *TypeEnv* are maps from names to *TypeExp*. Typing judgments of the form $\Gamma \vdash E : \tau$ relate a type environment (Γ), an expression (E), and a type expression (τ).

Boolean-valued expressions are called *atomic formula*:

$$\mathit{AtomicFormula} \overset{\text{def}}{=} \{E \mid E \in \mathit{Exp}, \exists \Gamma \in \mathit{TypeEnv}. \Gamma \vdash E : \mathit{Boolean}\}.$$

Since some atomic formulas may relate a pre-state to a post-state, the meaning function for expressions that type check in a type environment Γ, $\mathcal{E}_\Gamma$, takes two states as an argument. For a given type environment Γ, Γ-*state*s s and s' are states that agree with the type environment Γ in the sense that for each expression E, if $\Gamma \vdash E : \tau$, then $\mathcal{E}_\Gamma[\![E]\!](s, s')$ is an element of the meaning of the type τ.

A *specification* follows the grammar shown in Figure 3.4.

$$\begin{array}{lll}
\textit{Specfication} & ::= & \textit{Case} \\
 & \mid & \textit{Case} \vee \textit{Specification} \\
 & \mid & (\texttt{pre}\ \textit{AtomicFormulas}) \wedge \textit{Specification} \\
\textit{Case} & ::= & \texttt{pre}\ \textit{AtomicFormulas};\ \texttt{rest}\ \textit{AtomicFormulas} \\
\textit{AtomicFormulas} & ::= & \textit{AtomicFormula} \\
 & \mid & \textit{AtomicFormula} \wedge \textit{AtomicFormulas}
\end{array}$$

Figure 3.4: The abstract syntax of specifications.

$$\begin{array}{lll}
\textit{Specfication} & ::= & \textit{Case} \\
 & \mid & \textit{Case} \vee \textit{Specification} \\
\textit{Case} & ::= & \texttt{pre}\ \textit{AtomicFormulas};\ \texttt{rest}\ \textit{AtomicFormulas} \\
\textit{AtomicFormulas} & ::= & \textit{AtomicFormula} \\
 & \mid & \textit{AtomicFormula} \wedge \textit{AtomicFormulas}
\end{array}$$

Figure 3.5: The abstract syntax of specification normal form.

We use the notation *cases* to extract the cases of a specification as a set.

$$\textit{cases} : \textit{Specification} \to \mathcal{P}(\textit{Case})$$
$$\textit{cases}(S) \overset{\text{def}}{=} \textbf{let}\ c_1 \vee \ldots \vee c_n = S\ \textbf{in}\ \{c_1, \ldots, c_n\}$$

Sometimes it will be convenient to use operators *pre* and *rest* to extract the precondition and other atomic formulas (e.g., the postcondition) from a case; they have types:

$$\textit{pre} : \textit{Case} \to \mathcal{P}(\textit{AtomicFormula})$$
$$\textit{rest} : \textit{Case} \to \mathcal{P}(\textit{AtomicFormula}).$$

These operations satisfy, for all $P, Q \in \mathcal{P}(\mathit{AtomicFormula})$, and $S \in \mathit{Specification}$,

$$\mathit{pre}(\texttt{\textbf{pre}} \bigwedge P;\ \texttt{\textbf{rest}} \bigwedge Q) = P \tag{3.6}$$

$$\mathit{rest}(\texttt{\textbf{pre}} \bigwedge P;\ \texttt{\textbf{rest}} \bigwedge Q) = Q \tag{3.7}$$

$$\mathit{pre}((\texttt{\textbf{pre}} \bigwedge P) \wedge S) = P \cup \left(\bigcup_{c \in \mathit{cases} S} \mathit{pre}(c) \right) \tag{3.8}$$

$$\mathit{rest}((\texttt{\textbf{pre}} \bigwedge P) \wedge S) = \bigcup_{c \in \mathit{cases} S} \mathit{rest}(c). \tag{3.9}$$

A restriction of the general form of specifications defines the output format of *sp* and thus the input format of FAR. This format does not allow preconditions to be combined with specifications using $\wedge$.

Definition 2 (Specification Normal Form (SNF))**.** A specification is in *specification normal form (SNF)* if it follows the grammar in Figure 3.5.

Connection with SP

The foundation of the connection between a specification languages and the abstract specifications described in Section 3 and the formulas produced by *sp* is straightforward, because *sp* produces formulas in disjunctive normal form in which the formulas corresponding to preconditions are all disjoint. To illustrate this connection, the simple program, `cmp`, shown in Figure 3.6a.

```
public class Cmp {


    //@ requires true;
    public int cmp(int a, int b){
        if (a < b) {
            return -1;
        } else {
            if (a > b) {
                return 1;
            }
            return 0;
        }
     }

}
```

(a) The code for a simple comparison function.

```
public normal_behavior
  requires true;
  {|
    requires (a < b);
    ensures \result == -1;
  also
  {|
    requires !(a < b);
    requires (a > b);
    ensures \result == 1;
  also
    requires !(a < b);
    requires !(a > b);
    ensures \result == 0;
|}
|}
```

(b) The specification computed by *sp* for `cmp`.

Figure 3.6: (a) The Java code for a simple comparison function. Using the standard definition of strongest postconditions, one obtains the postcondition shown in (b) for `cmp`.

The computation of $sp(\texttt{cmp}, \phi)$ produces the following formula:

$$\begin{aligned} &(a < b \wedge \phi \wedge \textbf{result} = -1) \\ &\vee\ ((a > b \wedge \neg(a < b) \wedge \phi \wedge \textbf{result} = 1) \\ &\vee\ (\neg(a < b) \wedge \neg(a > b) \wedge \phi \wedge \textbf{result} = 0)) \end{aligned} \tag{3.10}$$

Formula (3.10) corresponds to the following abstract specification.

$$\begin{aligned} &(\textbf{pre}\ a < b \wedge \phi;\ \textbf{rest result} = -1) \\ &\vee\ (\textbf{pre}\ a > b \wedge \neg(a < b) \wedge \phi;\ \textbf{rest result} = 1) \\ &\vee\ (\textbf{pre}\ \neg(a < b) \wedge \neg(a > b) \wedge \phi;\ \textbf{rest result} = 0) \end{aligned} \tag{3.11}$$

Consider the contract in Listing 3.6b. In this contract, although the symbols are different, we can see terms that correspond to the terms found in the result of *sp* in formula (3.10) and the corresponding abstract specification in formula (3.11). Specifically, note that the three parts of the JML specification that are separated by "**also**" in Listing 3.6b correspond to the three cases (separated by $\vee$) of the abstract specification in formula (3.11). The translation to JML is very direct in this way.

For specification languages that do not have JML's specification cases it is still possible to use the output of FAR to make the specifications shorter. This would be done by translating a formula like formula (3.11) into a single postcondition, using a translation function *Tr* defined on specifications and cases as follows:

$$
\begin{aligned}
&\mathit{Tr}(c_1 \vee \ldots \vee c_n) = \mathit{Tr}(c_1) \wedge \ldots \wedge \mathit{Tr}(c_n)\\
&\mathit{Tr}(\mathtt{\textbf{pre}}\, p;\ \mathtt{\textbf{rest}}\, q) = (\mathtt{\textbf{old}}(p) \implies q)\\
&\mathit{Tr}((\mathtt{\textbf{pre}}\, p) \wedge S) = (\mathtt{\textbf{old}}(p) \implies \mathit{Tr}(S))
\end{aligned}
$$

Note the use of $\wedge$ instead of $\vee$ in the produced postcondition to connect cases. If this were not done, then the postcondition would be trivially satisfied whenever one of the preconditions was false. (Also for the distributed form of specification cases $(\mathtt{\textbf{pre}}\, p) \wedge S$, note that $p \implies (p' \implies q)$ is equivalent to $p \wedge p' \implies q$, so the first precondition (p) distributes over the others as required.) For example, formula (3.11) would turn into the following postcondition

$$
\begin{aligned}
&(\mathtt{\textbf{old}}(a < b \wedge \phi) \implies (\mathtt{\textbf{result}} = -1))\\
&\wedge\ (\mathtt{\textbf{old}}(a > b \wedge \neg(a < b) \wedge \phi) \implies (\mathtt{\textbf{result}} = 1))\\
&\wedge\ (\mathtt{\textbf{old}}(\neg(a < b) \wedge \neg(a > b) \wedge \phi) \implies (\mathtt{\textbf{result}} = 0))
\end{aligned}
\tag{3.12}
$$

Algorithm 1: FAR

```
Input: S, a specification in SNF (Def. 2)
       ∼, a sound equivalence relation (Def. 3)
Output: A specification
begin
2   $V \longleftarrow cases(\mathsf{S})$ // (Sec. 3)
3   $Residual \longleftarrow$ EmptyGraph()
    // merge the overlapping state spaces of $\mathsf{S}$
5    repeat
6       $(G, W) \longleftarrow$ ToGraph(V) // (Alg. 2)
7       $Scc \longleftarrow$ StronglyConnectedComponent(G)
8       $(Residual, V) \longleftarrow$ MergeSCC(V,W,Scc,Residual,∼) // (Alg. 3)
    until $|Scc| = 0$
    // connect each unmerged vertex to the root of the specification
11  $Residual.E = Residual.E \cup \{(Residual.root, v) \mid v \in V\}$
12  $Residual.V = Residual.V \cup \{V\}$
    // convert the residual graph back to a specification
14  return ToSpecification(Residual, Residual.root) // (Alg. 4)
end
```

Description of FAR

The main pseudocode of FAR is given in Algorithm 1. In this section we will discuss the main parts of the algorithm as they relate to Algorithms 1, 3, 2, and 4 and Figure 3.3. The FAR algorithm has 4 main parts, described below.

Input Specification. The input to FAR comes directly from the specification produced by the predicate transformer semantics. Since the specification is produced by sp, the specification will be in SNF. These clauses are placed in a data structure suitable for a graph-based analysis. For our implementation, we used the JPaul program analysis library [90]. This process happens on Line 2 of Algorithm 1.

Flattening. With the cases of the specification cases collected, the next step is to flatten the cases.

```
Algorithm 2: ToGraph
Input: V, a set of specification cases
       $\sim$, a sound equivalence relation (Def. 3)
Output: A graph $G$ and a table of weights $W$
begin
    $G \longleftarrow$ EmptyGraph()
    for $(l, r) \in \mathsf{V} \times \mathsf{V}$ such that $l \neq r$ do
        // add $l$ and $r$ to the list of vertices
        $G.V \longleftarrow G.V \cup \{l\} \cup \{r\}$
        // compute the weight modulo $\sim$
        $W[l][r] \longleftarrow |\{(x, y) \in \mathit{pre}(l) \times \mathit{pre}(r) \mid x{\sim}y\}|$
        // connect $l \to r$ if they share any clauses in common
        if $W[l][r] > 0$ then
            $G.E \longleftarrow G.E \cup \{(l, r)\}$
        end
    end
    return $(G, W)$
end
```

This is done by taking each node of the input specification that is not a disjunction node and detaching it from the tree. This forms a disjoint forest of all the specifications cases. Once flattening is then complete, the algorithm computes weights using an equivalence relation on pairs of distinct vertices. Weighted edges are shown by the dashed arrows in Figure 3.3. On Line 7 of Algorithm 2 this equivalence relation is denoted by the symbol $\sim$ (see Section 11). For simplicity, our prototype concludes that two clauses are equivalent if they are lexically identical. We show the soundness of this choice in Corollary 1. In our experimental analysis (Section 4) we find that this choice works out well. Note however that the relation used for FAR need not be lexical similarity; indeed, more interesting results can be achieved by considering other relations such as logical equivalence.

State Finding. In the state finding phase (Lines 5-8 of Algorithm 1), the dashed arrows leaving each vertex are added together and converted to graph weights as can be seen in Figure 3.3. For example, examining vertices C_2 and C_3 we see two edges going from C_2 to C_3.

```
Algorithm 3: MergeSCC
Input: V, a set of specification cases (Sec. 3)
       W, a table of weights between edge pairs of Scc
       Scc, the strongly connected component, a set of graphs of the form $\langle V, E\rangle$
       Residual, a graph of the form $\langle V, E\rangle$
       $\sim$, a sound equivalence relation (Def. 3)
Output: A pair consisting of the Residual and remaining vertices V
begin
2   for $Component \in Scc$ such that $|Component.V| > 1$ do
3       $maxWeight \longleftarrow$ the maximum weight of $Component$ via W
4       $(L, R) \longleftarrow$ the vertex pair $\in Component.E$ with weight $maxWeight$
        // compute intersection modulo $\sim$
6       $common \longleftarrow \{l \mid (l, r) \in pre(L) \times pre(R)$ where $l{\sim}r\}$
        // remove the common vertexes
8       $R' \longleftarrow$ pre $(pre(R) \setminus \{r \mid (l, r) \in pre(L) \times pre(R)$ where $l{\sim}r\})$; rest $rest(R)$
9       $L' \longleftarrow$ pre $(pre(L) \setminus common)$; rest $rest(L)$
        // in the residual graph, create a conjunction node and connect it to $L$ and $R$
11      Residual.$E \longleftarrow$
         Residual.$E \cup (\{common\} \times \{L', R'\}) \cup \{($Residual.$root, common)\}$
12      Residual.$V \longleftarrow$ Residual.$V \cup \{L'\} \cup \{R'\}$
        // remove merged nodes
14      V $\longleftarrow$ V $\setminus (\{L\} \cup \{R\})$
    end
16  return (Residual, V)
end
```

This means in the state finding phase we will place an edge with weight 2 between these two vertices. The algorithm continues in this fashion until it produces a weighted graph. Next, compute the connected components of this graph. In Figure 3.3, this process produces Component 1 and Component 2. Then, for each connected component, select the pair of vertices with the largest weight (most similarity). In Figure 3.3 this produces the choice (C_6, C_7) for Component 1 and $(C_2, C3)$ for Component 2.

Recombining. For each selected pair of vertices from the State Finding step we perform the following procedure (Lines 4-12 of Algorithm 3):

Algorithm 4: ToSpecification

```
Input: Residual, a graph of the form $\langle V, E \rangle$
       Root, the root of the graph
Output: A specification (Def. 1)
begin
    $B \longleftarrow \emptyset$
    for each $v$ adjacent to Root do
        if $v$ is a leaf then
            $B \longleftarrow B \cup v$
        else
            $B \longleftarrow B \cup (v \wedge \texttt{ToSpecification}(\text{Residual}, v))$
        end
    end
    return $\bigvee B$
end
```

First, we create a conjunction node (a logical AND) and connect it to each vertex in the pair. We then modify each vertex in the pair in the following way. The right vertex becomes the conjunction of all of the clauses that were lexically equal in both vertices. The left vertex becomes the disjunction of the set of clauses that *were not* lexically equal in both vertices. In Figure 3.3 this operation is denoted by the $*$ next to the vertex name. Next, we examine the connected component to which the pair belongs. If any vertex is adjacent to any of the elements of the pair, we remove that edge from the graph. This process is repeated for each connected component produced in the previous step. Lastly, we create a "super node", which will serve as the new root of the specification. This node should be a disjunction (logical OR) and be connected to the newly created conjunction nodes (the merged states in Figure 3.3). In the example in Figure 3.3, this leaves us with vertices C_1 and C_5. This subgraph is then fed back into the FAR algorithm until there are no remaining vertices. Completely disjoint vertices (those with nothing in common with any of the other vertices) form a disjoint forest in the final iteration of the algorithm. Once this outcome is reached, each vertex in the forest is connected unmodified to the super node. This resulting tree is then converted back into a specification.

Base Analysis Types

After an execution trace of a particular method is obtained, we next need to transform the trace and the underlying predicate AST into a practical contract. The portion of Strongarm that is responsible for this task is the various analyses in the contract transformation pipeline (see Figure 3.3). These base analysis types include standard tasks such as translation from internal variable representations to externalized (source code level) representations, removal of tautologies, determination of purity, and inference of frame axioms. For space considerations we omit a detailed description of these standard techniques.

Soundness of $\sim$

The FAR algorithm uses an equivalence relation $\sim$ to compute the intersection between two sets of preconditions. However, not just any equivalence relation on atomic formulas will do; the relation must preserve the meaning of atomic formulas.

Definition 3 (Sound Equivalence Relation)**.** An equivalence relation $\sim$ on atomic formulas is *sound* iff for all type environments Γ and all atomic formulas F and G such that $\Gamma \vdash F$: *Boolean* and $\Gamma \vdash G$: *Boolean*:

$$F \sim G \implies \forall\Gamma\text{-states } s, s' \,.\, \mathcal{E}_\Gamma[\![F]\!](s, s') = \mathcal{E}_\Gamma[\![G]\!](s, s').$$

The FAR algorithm works correctly if $\sim$ is any sound equivalence relation on atomic formulas. However, in this paper and our implementation we use lexical equivalence (i.e., textual identity) as the relation, hence we need the following corollary, whose proof is trivial.

Corollary 1 (Soundness of Lexical Equivalence)**.** *Lexical equivalence of atomic formulas is a* *sound equivalence relation.*

Soundness of FAR

In this section we show the soundness of FAR. Since soundness for FAR is defined with respect to satisfaction by programs (method bodies), we first define satisfaction for programs. In essence the following definition says that a program C satisfies a specification S if for every pre-state s that satisfies some case c's precondition, the semantics of C is such that the post-state s' that C produces (when run on s) satisfies c's postcondition.

Definition 4 (Satisfies a Specification)**.** Let Γ be a type environment. Let S be a specification in which each atomic formula contained in S type checks in Γ. Let C be a program whose semantics is captured by a state transformer χ that transforms Γ-states to Γ-states. Then C *satisfies* S, written $C \models_\Gamma S$, if and only if for all Γ-states s such that there is a case $c \in \textit{cases}(S)$ such that for each atomic formula $p \in \textit{pre}(c)$, $\mathcal{E}_\Gamma[\![p]\!](s, s)$ is true, and $\chi(s) = s'$, then for each atomic formula $q \in \textit{rest}(c)$, $\mathcal{E}_\Gamma[\![q]\!](s, s')$ is true.

The following lemma relates to the invariant of the FAR algorithm that we use to prove its soundness.

Lemma 1 (`ToSpecification` on Flat Tree)**.** *Let G be a graph $\langle V, E\rangle$ with root $G.root$ such that $\forall v \in V\ \langle G.root, v\rangle \in E$. Then* `ToSpecification`$(G, G.root)$ *produces the disjunction of each $v \in V$.*

Proof. Let G be a graph $\langle V, E\rangle$ with root $G.root$ such that $\forall v \in V\ \langle G.root, v\rangle \in E$. On Line 3, `ToSpecification` iterates over each vertex in G. On Line 4, we test if v is a leaf. Since all v are leaves, on return on Line 10, B contains the set V. Since $\bigvee B$ produces the disjunction of every element of B, `ToSpecification`$(G, G.root)$ produces the disjunction of each $v \in V$. $\square$

The following corollary also relates to the invariant in the FAR algorithm.

Corollary 2 (Two-Leaf Pairs Converted to Specifications)**.** *Let G be a graph $\langle V, E\rangle$ with root $G.root$. For any pair of leaf vertices (l, r) adjacent to $G.root$,* `ToSpecification`$(G, G.root)$ *contains a disjunction of l and r if and only if $\{l, r\} \subseteq V$.*

The following definition of soundness for FAR is reasonable because if FAR does not change the meaning of a specification, then it is only simplifying it. Furthermore, since we pass specifications to FAR that are derived from *sp* and these capture the behavior of the code, if FAR does not change the meaning of these specifications, then they still capture the behavior of the code.

Definition 5 (Soundness of FAR)**.** Let $\sim$ be a sound equivalence relation on atomic formulas. Then *FAR is correct with respect to $\sim$* iff for all programs C and for all specifications S:

$$C \models_\Gamma S \iff C \models_\Gamma FAR(S, \sim).$$

Theorem 1 (FAR is Sound)**.** *If $\sim$ is a sound equivalence relation, then FAR is correct with respect to $\sim$.*

Proof. Assume $\sim$ is a sound equivalence relation. Let Γ be a type environment. Let C be a command with semantics χ that takes Γ-states to Γ-states. Let S be a specification in SNF in which each atomic formula contained in S typechecks in Γ.

The proof uses a loop invariant maintained by FAR. FAR maintains the state of its execution according to the tuple $\langle S, V, Residual\rangle$, where V is a list of unmerged vertices and $Residual$ is the residual graph formed from previous $MergeSCC$ calls. We refer to the state $\langle S, V, Residual\rangle$ as FAR_{State} and define the following invariant that operates on its elements:

$$\textit{inv} : \textit{Specification} \rightarrow \{Case\} \rightarrow Graph \rightarrow \textit{Boolean}$$

$$\textit{inv}(S, V, G) \overset{\text{def}}{=} \textbf{let } G' = \langle G.V \cup V, G.E \cup \{(G.root, v) \mid v \in V\}\rangle \textbf{ in}$$
$$C \models_\Gamma S \iff C \models_\Gamma FAR(\texttt{ToSpecification}(G', G'.root), \sim)$$

The graph G' formed in *inv* by $\langle G.V \cup V, G.E \cup \{(G.root, v) \mid v \in V\}\rangle$ encapsulates Lines 11 and 12 of FAR which are responsible for taking the remaining elements of V and adding them to the residual graph. As can be seen on Line 11 they are added to the root of $Residual$. Encapsulating the end of the FAR procedure in this way demonstrates what a snapshot of the state of FAR would be for any given iteration of the loop on Line 5.

Establishing. The invariant *inv* is established after the execution on the statement on Line 3 of Algorithm 1. At this point $FAR_{State} = \langle S, \textit{cases}(S), \langle\emptyset, \emptyset\rangle\rangle$. In *inv*, since G is empty, G' is a graph with a root that is attached via an edge to each leaf in V. By Lemma 1, we know that $\texttt{ToSpecification}(G', G'.root)$ is the disjunction of all of the cases of V. While this may reorder the cases, the semantics of a specification is unaffected by the order of the cases, therefore the invariant is established.

Maintaining. Lines 6 and 7 have no effect on the values of FAR_{State} and therefore do not directly affect the value of *inv*. We are concerned with the effect of the computation on Line 8. The result of Line 8 is impacted by the value of V, the value of which takes on several cases.

Case 1 Consider $V = \emptyset$. In this case $\texttt{MergeSCC}(V, W, Scc, Residual, \sim) = \langle V, Residual\rangle$. That is, $\langle V, Residual\rangle$ is not modified. This case is trivial since the connected component of an empty graph is empty as well. In this case $\langle V, Residual\rangle$ is not modified and hence FAR_{State} remains the same.

Case 2 Consider $V \neq \emptyset$ and for every pair $(l, r) \in V \times V$ such that $l \neq r$ the formula $|\{(x, y) \in \textit{pre}(l) \times \textit{pre}(r) \mid x \sim y\}| = 0$ is true. In this case, for each iteration of the loop of Algorithm 2, the condition on Line 9 is false. This case is equivalent to case 1, since there are no edges and therefore the graph is a disjoint forest. In this case, the connected components all contain one vertex. However, since the loop on Line 2 of Algorithm 3 only executes in the event $|Component.V| > 1$, the pair $\langle V, Residual\rangle$ is not modified and hence FAR_{State} remains the same.

Case 3 Consider $V \neq \emptyset$ and there exists some pair $(l, r) \in V \times V$ such that $l \neq r$ where $|\{(x, y) \in \textit{pre}(l) \times \textit{pre}(r) \mid x \sim y\}| > 0$. In this case, at least one component of Scc will satisfy the requirement $|Component.V| > 1$ of the loop on line 2 of Algorithm 3. We consider the effect of a single component on the value of *inv*. Let the pair selected on Line 4 be the pair (L, R). Let *common* be the set $\{l \mid (l, r) \in \textit{pre}(L) \times \textit{pre}(R) \text{ where } l \sim r\}$ computed on Line 6 of Algorithm 3. On exit from Line 14 of Algorithm 3 $FAR_{State} = \langle S, V', Residual'\rangle$ where the following updates have been made:

$V' = V \setminus (\{R\} \cup \{L\}) \implies$ `ToSpecification`$(Residual, Residual.root)$ does not contain $L \vee R$ (by Corollary 2), which causes *inv* to no longer hold.

However, the following computation causes *inv* to hold again:

$Residual'.E = Residual.E \cup (\{\textit{common}\} \times \{L', R'\}) \cup \{(Residual.root, \textit{common})\}$

$Residual'.V = Residual.V \cup \{L\} \cup \{R\}$

where

$L' = \textbf{pre}\ (\textit{pre}(L) \setminus \textit{common});\ \textbf{rest}\ \textit{rest}(L)$

$R' = \textbf{pre}\ (\textit{pre}(R) \setminus \{r \mid (l, r) \in \textit{pre}(L) \times \textit{pre}(R) \text{ where } l \sim r\});\ \textbf{rest}\ \textit{rest}(R)$

On Line 14, L and R are removed from V. Prior to removal, an evaluation of *inv* would produce a disjunction of L and R from Corollary 2 in the final specification. They are replaced in $Residual$ by something that translates to an equivalent specification. That is, the formula $(L \vee R)$ is replaced by $(common \wedge (L' \vee R'))$ on Lines 11 and 12 when $Residual$ is updated. Since $\sim$ is a sound equivalence relation, $(L \vee R) \equiv (common \wedge (L' \vee R')) \implies$ *inv*$(S, V, Residual)$ and therefore the invariant *inv* is maintained.

Terminating. In the case evaluation in the maintaining step, we showed how *inv* holds at each step of the loop in FAR. Since Lines 11 and 12 are encapsulated by *inv*, we know that *inv* holds on exit from FAR which gives us the desired result. □

CHAPTER 4: A TECHNICAL EVALUATION OF FAR AND STRONGARM

Previous attempts to infer specifications in other works have resulted in specifications that were either too short to be useful, required the presence of tests suites (in the case of runtime inference), or have produced specifications that were too long to be easily understandable. One of the goals of this work is to improve on previous attempts, in this chapter we examine our claims with an experimental evaluation of our work.

In this chapter we evaluate the performance of Strongarm on the task of inferring specifications. In the next section we explain our experimental setup and provide specific details about the source code we conducted our experiments on. Following this, our evaluation looks at the performance of Strongarm's inference from the following five perspectives:

1. **Effectiveness of Inference** How many of the candidate methods we were able to infer?
2. **Efficiency of Reduction** By how much were the specifications reduced?
3. **Complexity of Inferred Specifications** Overly complex and overly simplistic specifications are not practical. What are the characteristics of the inferred specifications?
4. **Performance of Inference Procedure** How well (with respect to time) did Strongarm perform at the task of inferring specifications?
5. **Performance of FAR** How effective is FAR at reducing specification nesting, decreasing specification length, decreasing prover execution time, and decreasing proof length?

Finally, in Section 4 we conclude with a human evaluation of the inferred specifications produced by Strongarm.

Evaluation Methodology

We designed a series of experiments designed to examine the effectiveness of Strongarm at inferring practical specifications. We selected a cross section of popular Java libraries: JUnit4 (JU4), JSON-Java (JJA), Commons-CSV (CSV), Commons-CLI (CLI), Commons-Codec (COD), Commons-Email (EMA), and Commons-IO (CIO). Code metrics pertaining to these libraries are summarized in Table 4.1.

Table 4.1: Code metrics for APIs used in evaluation.

API Name	SLOC	Methods	Files	Version
JUnit4	10,018	1,230	193	4.13
JSON-Java	3,201	200	18	20160212
Commons-CSV	1,501	158	10	1.4
Commons-CLI	2,666	194	22	1.3.1
Commons-Codec	6,607	509	60	1.10
Commons-Email	2,734	192	22	1.4
Commons-IO	9,836	955	115	2.5
Total	**36,563**	**2,331**	**440**	-

A method for inferring preconditions in large corpora was recently investigated in the work of Nguyen et al. [72]. However, rather than specifying the preconditions for the methods in our study, we assume a vacuously true precondition, namely `true`. Prior to construction of the final specification, the default precondition is removed and not tabulated in the final specification analysis. We do this so as to not simultaneously test the results of our work in parallel with the technique of Nguyen. As noted in Section 4, we do not specifically attempt to infer specifications for loops and instead rely on user-written loop conditions. For the study presented in this paper loops were not annotated with such conditions. Additionally, Strongarm's inference technique assumes any field referenced in the body of a method should be visible in the inferred specification. However, in JML this is considered an error by default. For this reason, all private fields referenced in specifications have been given the special annotation `spec_public` which allows private fields to appear in specifications. This promotion of private fields to `spec_public` is performed automatically by Strongarm during inference and reflected in the inferred specifications; future work includes using JML features such as model fields to avoid declaring all fields to be `spec_public`. Additionally, during our evaluation we discovered there were several methods we were unable to validate due to bugs in OpenJML; these methods were explicitly skipped in our evaluation.

Experiments were performed on the Stokes HPC cluster[1] at the University of Central Florida. Each job node was configured with 6 Intel Xeon 64-bit processors, 42GB of RAM, and used Oracle JDK 1.8.0.131. Strongarm produces extensive telemetry data as well as the inferred specifications. The data presented in this section is based on mining this telemetry data.

Verification of Inferred Specifications

Once a specification is inferred, we must have a standard way of knowing if the specification itself is correct. In our experiments we validated the inferred specifications in three different ways. First, in the creation of Strongarm, we built a comprehensive test suite consisting of approximately 100 hand written and hand verified test cases. Strongarm passes this test suite. Second, once inferred specifications are produced, we use OpenJML's built in support for type checking to type check the produced specifications. All of the specifications produced by Strongarm type check. Lastly, we use OpenJML's Extended Static Checker (with Z3 [25] version 4.3.0), and all of the specifications produced verify. Using this technique we were able to verify 70% of our inferred specifications. In practice Strongarm would check these specifications before submitting them to a user and therefore would not produce an invalid specification. The cases we were unable to verify

[1]https://arcc.ist.ucf.edu/index.php/resources/stokes/about-stokes

were caused by tool error or implementation issues in Strongarm. Verification in this fashion also checks for unsatisfiable clauses, which should not be present in practical specifications. In doing this, as noted in Section 4, we do not consider the exceptional behavior of the method in question.

Threats to Validity

As mentioned in Section 4, we use the OpenJML tool to check the results of our inferred specifications. A positive result from OpenJML certifies that the program code satisfies the given specification. This is an especially strong guarantee; since checking is done statically, this certifies that *for all runs* (and potential input values) the specifications are valid. However, this depends on the soundness of the tool. This is a limitation in the following ways. First, OpenJML itself might have bugs and therefore may certify programs that are not correct. Second, the theory behind OpenJML itself might not be sound, i.e., it may admit incorrect programs (programs that do not satisfy their specification) under certain circumstances; however, it is believed that this second problem is limited to JML features with semantics that are not quite settled yet (such as details concerning invariants). Our study is not impacted by known problems in the semantics of JML. Lastly, in choosing JML as our target specification language, our approach to specification inference may not generalize to other specification languages as well as it has with JML. However, there are many Hoare-style specification languages that use pre- and postcondition specifications, such as Eiffel, and retargeting Strongarm to these specification languages should be a straightforward exercise for future work.

Limitations

In designing Strongarm, we intentionally made some trade-offs to simplify its implementation. The two major limitations of Strongarm are seen in our handling of exceptions and in our handling of loop constructs. First, although JML allows for descriptions of exceptional method behavior, in Strongarm we do not attempt to infer this behavior (although information about the exceptional behavior of codes is present in our AST; it is simply elided). Instead this task is relegated to future work. Second, as a simplifying assumption, in our implementation we assume that in the presence of loops that Strongarm will always have access to expert-written loop invariants. From these loop invariants we apply the standard Hoare loop rules to facilitate inferring postconditions.

Effectiveness of Inference

For each of the libraries we categorized the status of an inference attempt in one of four different categories. We give an explanation of the categories in this section as well as provide some analysis of our findings; see Figure 4.1.

The status *Inferred* indicates that Strongarm successfully inferred a specification for a method. We define success as containing at least one postcondition in the form of an `ensures` or `assignable` clause (in the case a frame axiom is necessary). Other clauses may be (and often are) present. For all experiments Strongarm succeeded in producing a specification more than 74.0% of the time (overall), and in its worst performance produced a specification 48.1% (in Commons-CSV). Strongarm's best performance in terms of succeeding in producing a specification was found in our evaluation of JUnit4 (92.5%). This was largely due to the fact that the control flow graphs resulting from converting the JUnit4 library to the basic block format resulting in significantly smaller control flow graphs compared to other libraries.

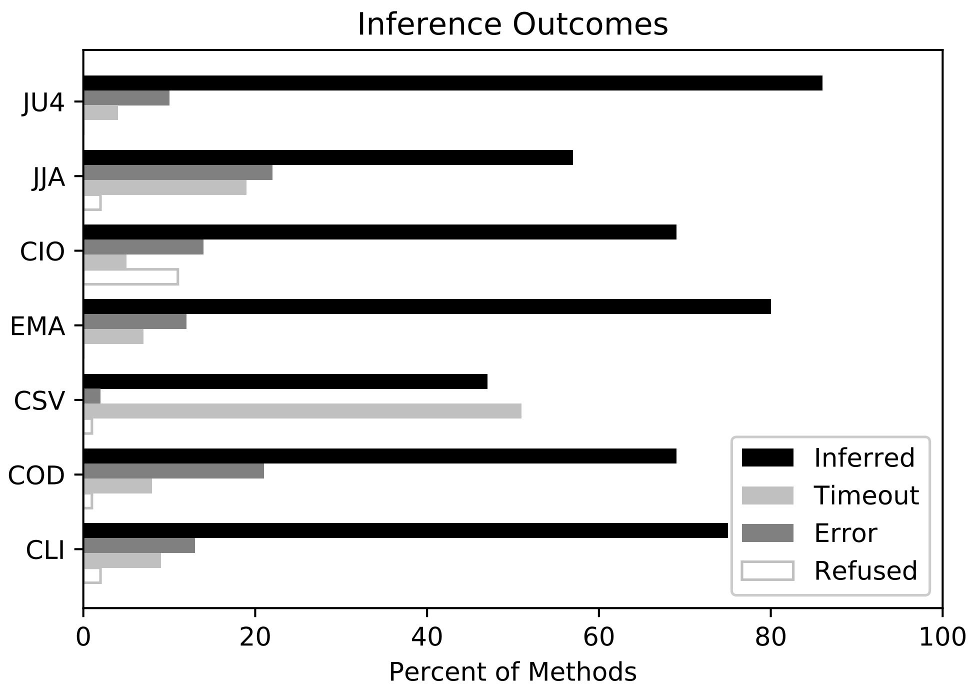


Figure 4.1: A summary of inference outcomes for Strongarm on our 7 test libraries.

A status of *Timeout* indicates that the inference process was aborted before inference could complete. For this paper we used a timeout of 300 seconds (5 minutes). We determined this timeout through repeated experimentation with different timeouts ranging up to 20 minutes. In our initial tests we determined that inference attempts that did not complete within 5 minutes were not able to complete in 20 minutes either. Higher settings for timeout timeouts may reduce the number of specifications that fail to be inferred. However, in our study we found that the combined timeout was only $\approx 9\%$. When a timeout occurs the intermediate results are discarded and not included in our remaining analyses. In the results reported in Figure 4.1 we can see that the two worst performing subjects in terms of timeout are Commons-CSV (51.3%) and JSON-Java (23.0%). Unlike the other test candidates, both Commons-CSV and JSON-Java are both parsers that contain deeply nested code. In all other libraries timeout performance was excellent and perhaps these results suggest that inferring specifications for parsers, since they are inherently very recursive.

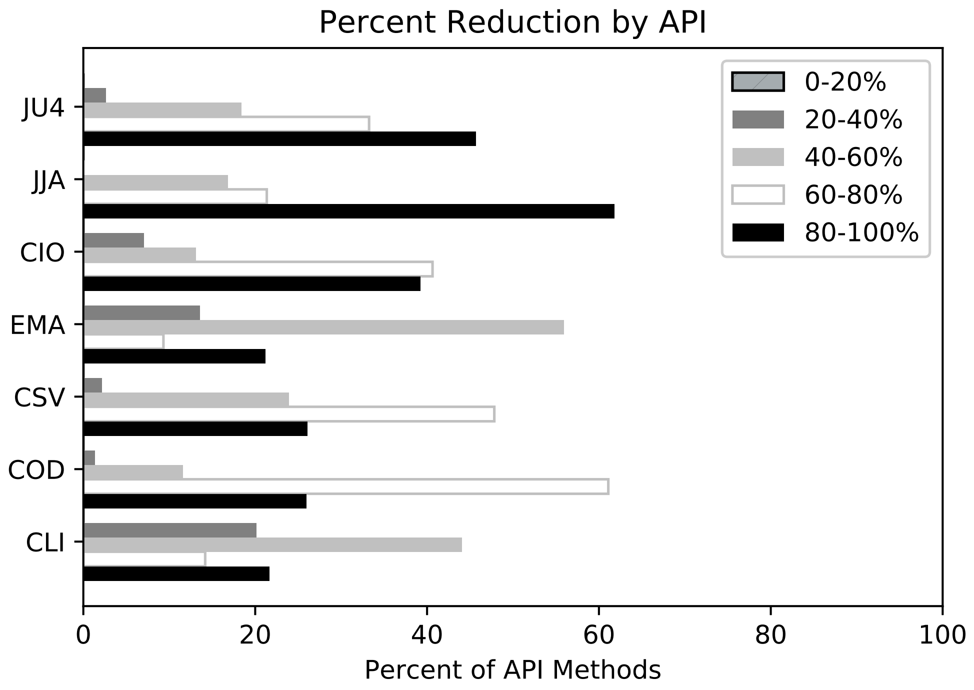


Figure 4.2: A summary of specification length reduction for Strongarm on our 7 test libraries.

A status of *Refused* means that Strongarm did not attempt to infer a specification, because the control flow graph (CFG) size was larger than a preset limit (a CFG size of 500 nodes). Similar to the timeout parameter, larger CFGs typically take much longer to infer. In our evaluation a size of 500 proved to be a reasonable choice, since the number of refused methods represented only 3.9% overall. This number is partially inflated by the unusually high number of refused methods in Commons-IO. This is explained by higher CFG complexity relative to the other libraries in the test suite. This additional complexity comes from the way the verification conditions for exceptional code are generated. Although we are not inferring the specifications for the exceptional specifications cases, the exceptional information exists in the CFG that Strongarm analyzes to infer the normal specification cases. Since Commons-IO deals with input/output related functions it has an unusually high number of exceptions. This greatly inflates the size of the CFG, which made fewer of its methods usable for our study.

This effect could be mitigated if the exceptional nodes were removed from the CFG prior to inference but such a change would make it then impossible to later infer the exceptional behavior of methods.

A status of *Error* means that Strongarm encountered an internal error during inference. We manually investigated these errors and found them to be generated from current limitations in OpenJML itself. For example, certain features, such as enumerated types, are not currently supported in OpenJML (although they are valid in JML itself). Our implementation is based on OpenJML 0.8.12; we expect to be able to reduce the amount of internal error our tool encounters as errors are corrected in OpenJML.

Efficiency of Reduction

As discussed in Sections 2 and 3, one of the problems impacting specification inference by symbolic execution (and therefore predicate transformers) is the length of the resulting inferred specification. In this section we will evaluate how effective Strongarm was at reducing the length of specifications inferred by symbolic execution.

In Figure 4.3, we show the length of the initial and final inferred specifications in terms of lines of code. Additionally, since short source code length can be a desirable quality from a software engineering and code readability perspective, we also provide two reference lines: one at one full page (80 lines) and one at a quarter page (20 lines). In our analysis, 95.0% of specifications fell below one full page length and 84.6% fell below one quarter page length. Using the page length lines as a proxy for specification for practicality, this information suggests that Strongarm's techniques for reducing specification size were effective in reducing the size of most of the inferred specifications.

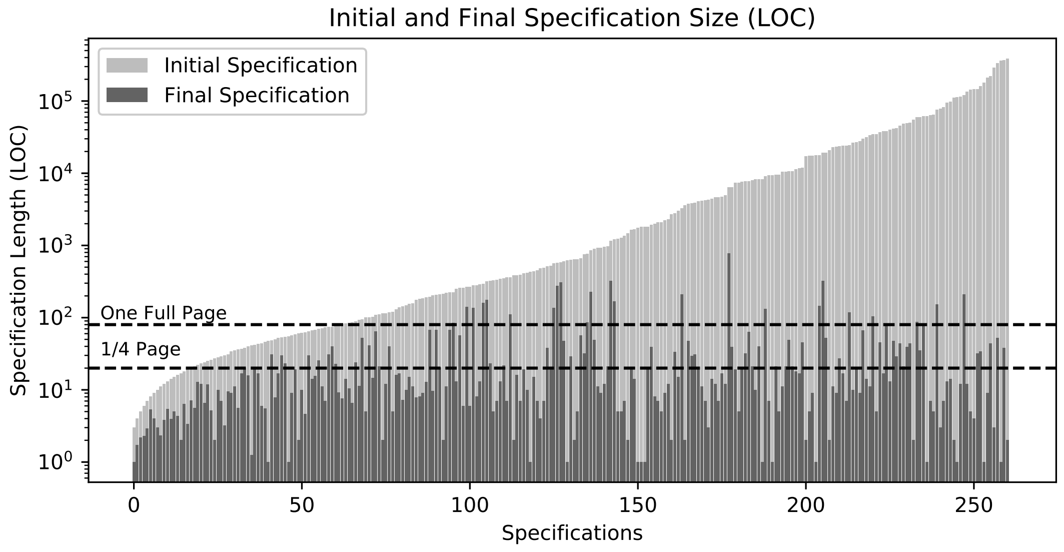


Figure 4.3: An analysis of initial and final specification length for all inferred specifications (all test libraries combined, binned to 262 equal width bins). The length of the initial specification following symbolic execution is denoted by the light lines in the above figure. The dark lines denote the specification length after Strongarm's specification reduction techniques are applied. For the purposes of this paper we consider 1/4 page to be equal to 20 lines.

The results in Figure 4.3 give an overview of all inferred specifications. However, Strongarm's effectiveness at reducing the final specification length of individual code bases varied significantly. In Figure 4.2, we can see a breakdown of Strongarm's effectiveness at reducing the final size of inferred specifications. Of the most interest are 80-100% category (the *most* reduction), and the 20-40% reduction category (the *least* reduction). Although we created a category for it, none of the code bases contained specifications that were reduced in the 0-20% category. In our experiment, we found that the 80-100% category contained 41.6% of all methods. In Figure 4.2 we see that the worst performer (least reduction) was Commons-Email. We did further analysis of this phenomenon where we looked at the length of the methods being inferred, the control flow depth, and the resulting reduction classification. We did not manage to find a relationship between these variables, which may suggest that these contracts failed to reduce more significantly because they described essential (non-reducible) logical states rather than non-practical artifacts as described in Section 3.

Complexity of Inferred Specifications

Criticisms of prior work on specification inference have included that the inferred specifications were either too short and not descriptive enough or template based and not expressive enough to capture the meaning of programs not falling within the parameters of the templates. We examine complexity in three different ways. First, in Figure 4.3 we give metrics on the variation in the length of specifications produced by Strongarm. To quantify the distribution of types of clauses Strongarm was able to infer, in Table 4.2, we report on the variation of clauses present in the specifications inferred by Strongarm for each of our target codebases. In Table 4.2 we can see that inferred specifications were comprised of mostly `requires` clauses, followed by roughly 4 times less `ensures` clauses as well as smaller numbers of `pure` and `assignable` clauses. Strongarm is able to handle quantifiers such as `forall`, however we did not examine it's distribution in the inferred specifications. In Figure 4.5, we detail the specification case nesting present in the inferred specifications. This is discussed further in Section 4.

Performance of Inference

To better understand the runtime performance characteristics of Strongarm we collected telemetry data about its performance.

Table 4.2: Summary of JML clauses in inferred specifications.

API	Methods	`requires`	`ensures`	`assignable`	`pure`
JU4	1,230	845	631	153	588
JJA	200	209	183	24	40
CSV	158	70	91	34	23
CLI	194	1,301	422	118	27
COD	509	1,533	569	105	90
EMA	192	7,139	622	191	47
CIO	955	1,483	681	270	641
Total	**2,331**	**12,580**	**3,199**	**895**	**1,456**
Ratio (Clause:Method)	-	5.39	1.37	0.38	0.62

In Figure 4.4, we can see that the data is tightly clustered on the y axis between 10^1 and 10^2 ms. Note that inference time in our experiment was limited to 5 minutes (3×10^5 ms). Many of the observed data points fell within this region, suggesting a linear fit with the exception of some very large control flow graphs which fell in the region bordering the timeout. These data points were identified as belonging mostly to Commons-IO and due largely to the number of exceptional flows present in the resulting control flow graphs (see Section 4 for more details). As discussed in Section 4, using the timeout threshold of 5 minutes with the current performance characteristics allowed us to infer more than 75% of the methods we considered for this study with an average inference time of just 2 seconds per method.

Performance of FAR

To better understand the performance of FAR with respect to more standard techniques such as removing tautologies, we conducted further evaluation to study FAR.

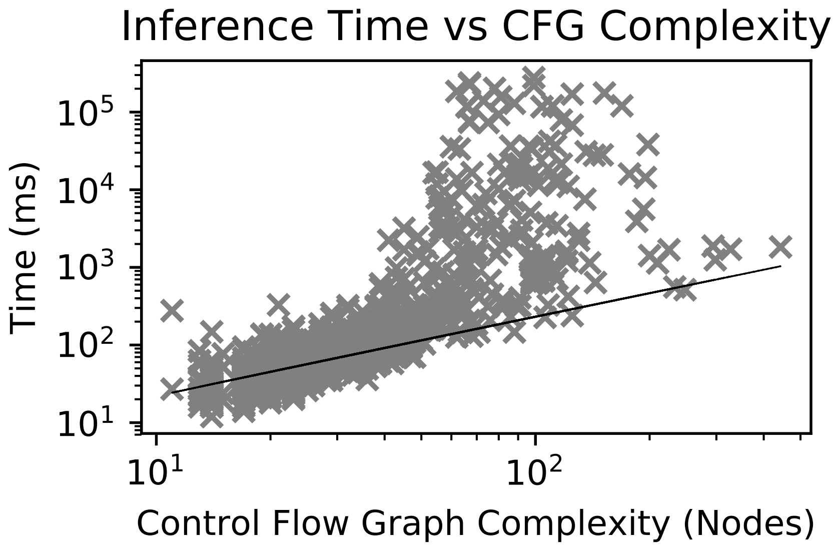


Figure 4.4: The performance of Strongarm's inference pipeline given in terms of the control flow graph complexity.

We conducted our study by taking the same 7 libraries used throughout this section and examining the effect of running all of the base analysis types (Section 11) with the exception of FAR in one run and all of the analysis types *with* FAR added back into the analysis pipeline. To understand the characteristics of specifications before and after FAR is applied to them we observed 4 different metrics. The first metric we observed was **specification nesting**. Specification nesting is defined as the lexical depth of a specification. In Figure 4.5, we present the performance of FAR in reducing nesting of specifications. In this case of all libraries, FAR achieved a reduction in nesting, producing an overall average reduction of 73.8%.

In our examination of the **percent reduction** aspects of FAR, we found FAR's effect on specifications is not necessarily to produce large reductions in the line length of specifications; FAR's contribution to line length reduction is small compared to the contribution of the other steps (FAR contributes an additional $\approx 10\%$ reduction overall).

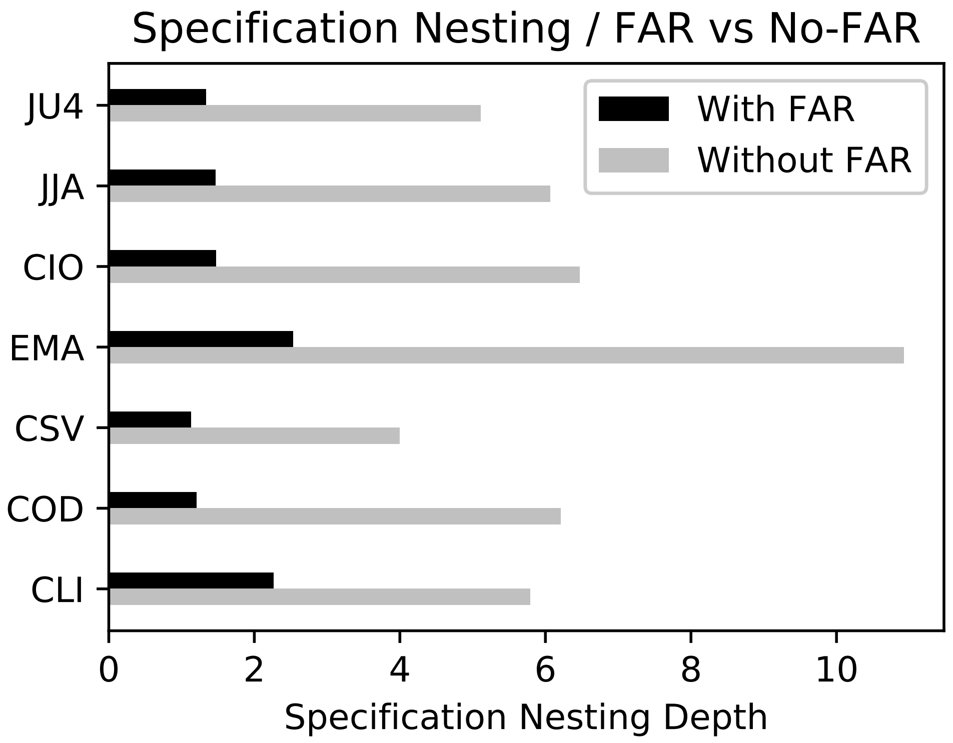
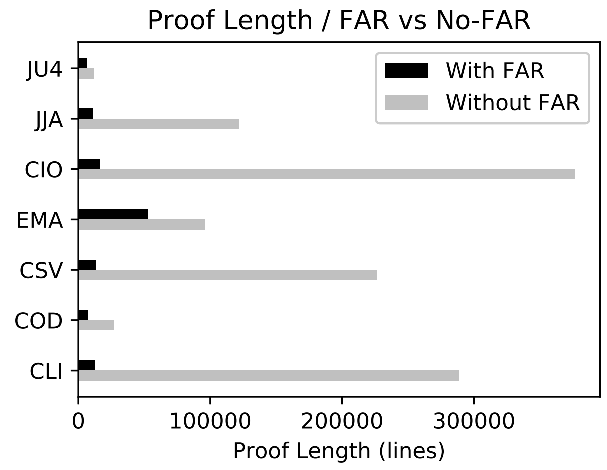


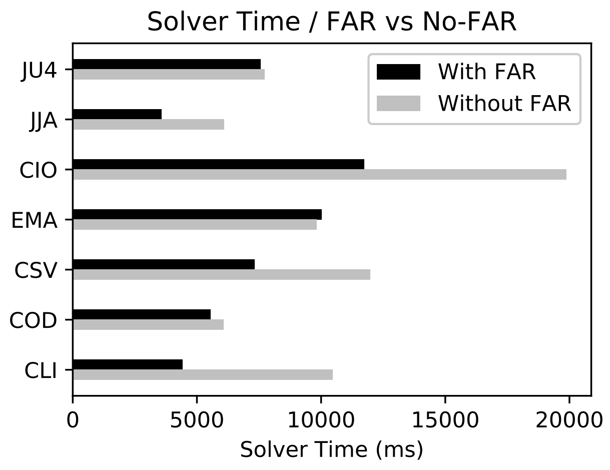
Figure 4.5: The effect of FAR on the nesting of inferred specifications

Ultimately, for a specification to be most useful, it should be used to verify the implementation of the code it specifies. In Figure 4.6a, we examine the impact of FAR on the **proof length**, i.e., the length of the SMT conditions generated that are needed to verify the correctness of a specification in relation to its implementation. In Figure 4.6a we can see that FAR has a large impact on the size of the generated SMT proofs for the code samples we studied. Overall, *FAR reduces the size of the generated SMT conditions by 76.7%.*

In the previous paragraph we saw that FAR is effective for reducing the size of the proofs required to verify implementations of inferred specifications. To determine the impact of the reduced length on the time to run the SMT solver on these proofs, we studied the **solver times** with and without FAR.



(a)



(b)

Figure 4.6: The effect FAR on (a) the length of the SMT conditions generated from specifications (b) the time needed to check the inferred specifications.

In Figure 4.6b we show the time taken to prove the specifications inferred with and without FAR. As can be seen in all cases, FAR reduces the time taken to verify the SMT conditions. Across all libraries, FAR reduces the prover execution time by 26.7%.

A Study of the Inferred Specifications

To determine if the specifications produced by Strongarm were useful to human readers, we conducted a small study to examine their usefulness. In general, we believe that practical specifications are more useful for human readers than those that are not, since they tend to be shorter and more to the point. However, we defined additional criteria against which we designed our study.

Namely, we selected the following criteria and say that a specification is *practial* if they:

- do not contain redundant formulas,

- do not contain unsatisfiable formulas,
- do not contain tautological formulas,
- specify frame axioms [13]),
- specify when a method is pure (has no side-effects), and
- only use names that are visible to a method's clients.

Strongarm produces specifications that do not have these problems. However, our tool has been constructed in a way that allows us to selectively disable the optimizations that remove these problems. This allowed us to design a study of the effect of these optimizations on the usability of the inferred specification.

Methodology. Using the specifications inferred by Strongarm, we conducted a survey of people familiar with the Java Modeling Language (JML) [55]. The survey used 12 pairs of method specifications inferred by Strongarm; to highlight the aspects of our definition of practical specifications, one element of each pair had one aspect, e.g., removal or unsatisfiable formulas, disabled. The survey was completed by 25 people, 18 of which said that they "definitely" had experience reading JML and 7 of which had some experience with JML. All correctly answered a simple question about JML that tested their understanding of a method specification with two specification cases. (A method *specification case* in JML is a pre- and postcondition specification, with optional frame axioms, that must be satisfied whenever the case's precondition is true when the method is called.) The questions in the survey were all simplifications of pairs of output given by our tool, one of which was not processed to remove a single impractical feature (according to the above definition).

Results. An unsatisfiable precondition, such as `!true`, causes the specification case it appears in to be useless. 87.5% of the survey respondents preferred a specification without unsatisfiable preconditions, including preconditions that required non-null fields to be null. Another example of unsatisfiable specifications comes when a specification case has two mutually-contradictory clauses; in this case all respondents preferred a specification without such unsatisfiable combinations of clauses (75% of them strongly so).

According to the survey, 95% of the respondents preferred a specification without the tautological postcondition `true == true`. Also, 62.5% of the respondents preferred a specification without tautologies such as `requires true` and more subtle tautologies involving non-null declarations. Another 87.5% preferred a specification without subtle redundancies such as `"No resource defined"!= null`. In another question, 80% of the respondents preferred a specification without redundant clauses requiring non-null fields to be not null. In another question, 93.3% of the respondents preferred a specification without duplicated specification cases.

Regarding frame axioms, the survey contrasted a specification without the frame `assignable this.name` with a nearly identical specification with the assignable clause. Although the specification without the frame axiom is shorter, 86.7% of the respondents preferred the specification with the frame axiom (53% strongly so).

In JML a pure method is one without any side-effects, and is specified by the `pure` keyword. The `pure` keyword also functions as a strong frame axiom. 66.7% of the respondents preferred a specification to one just like it but without the keyword `pure`.

Specifications with internally-generated variable names, such as `ASSERT_1320_6475` can be hard to follow. 57.9% of respondents preferred a specification that only used the `pure` annotation without requires and ensures clauses that mentioned such a variable name.

In sum, the majority of the survey respondents preferred specifications that satisfy our notion of a "practical" specification (the kind Strongarm produces), even if the differences were rather subtle.

Related Work

In addition to the work discussed in Section 3, there are several other systems and papers relevant to this paper. The closest related work to Strongarm is the Houdini system by Flanagan and Leino [33]. Houdini directly targets ESC/Java with the goal of statically inferring specifications for Java. Similarly, in our work we target OpenJML, a successor to ESC/Java. Rather than symbolically computing specifications as in Strongarm, Houdini instead applies templates, i.e., commonly found specification patterns; to search for a specification, Houdini tries all of these patterns and checks the candidate specifications using ESC/Java. In contrast to Strongarm, Houdini also attempts to infer object invariants using the same mechanism, which Strongarm does not attempt.

Ernst et al.'s work on Daikon [31] is also relevant. In contrast with Strongarm (and Houdini), Daikon is a runtime approach. To produce specifications, Daikon requires a test suite that calls the code that is being targeted for inference. Daikon thus relies on the existence of test code in order to function. Moreover, even when test code is available, dynamic techniques can fail since code coverage and branch coverage are typically not sufficient to produce specifications [45, 44] as many test suites focus on corner cases and are thus not suitable for specification inference [74].

Similarly, in their work on discovering relational specifications, Smith et al. discover specifications by looking at program outputs [88]. Our approach is fully static does not require running the code.

In their work on API usage error detection, Murali et al. investigate detecting API usage errors in applications by using Bayesian inference [67]. Our approach does not use machine learning to generate its specifications and therefore does not require training examples to infer specifications.

Our goal of inferring postconditions is also related to previous work in specification mining to infer preconditions by Nguyen et al. [72]. Nguyen et al. propose that one can infer the precondition of a method by examining the way it is used. Our approach is different in that Strongarm computes the specification based on the text of a method, not on code that calls it.

CHAPTER 5: A POLICY SPECIFICATION LANGUAGE FOR CONDITIONAL GRADUAL RELEASE

In Chapter 2 we discussed a system that facilitated specification authoring, sharing, and usage. Then in Chapter 3, to address the problem of lowering the cost of producing specifications we discussed a technique for inferring specifications automatically. Now we turn our attention to the challenges of designing a specification language for specifying information flow policies.

There are numerous design challenges that impact this goal. This is because, to be useful, information flow control policies need to specify declassifications and the conditions under which declassification can occur. However, attaching declassification operations to program statements makes the policies hard to find, which makes auditing difficult. Our policy specification language, *Evidently*, allows the specification of information flow control policies separately from a program, supporting conditional gradual releases that can be automatically enforced. By separating the policy from the program, scattering of policy details among the program's code is avoided, which should make policies easier to audit; also program changes can be limited to a policy module that describes program interactions, which should help policy specifications be more resilient to program changes.

Introduction

Auditing is an important dimension of an overall approach to security. However, in many approaches to specifying information flow control policies, the rules of the policy and when information can be declassified are interwoven with the text of the program, making the auditing process difficult. For example, the policies described by Chong and Myers [18] are written with declassification expressions in the code. Checking that such declassifications express the intended policy is important but requires a labor-intensive auditing step, in which the auditor synthesizes meaning from the declassification expressions scattered throughout the program. Moreover, when the program changes, all changed areas of the program must be examined to check that declassifications are used in a way consistent with the intended policy. Similar understanding and maintenance problems occur with other policy languages in which declassifications JIF [70], JFlow [69], JRif [53], and Paragon [14]. Some policy languages such as Jeeves [99] aim to address this problem. We provide a comparison of this work with *Evidently* in Section 5.

Our goal is to create a specification language that can describe policies about information flow (including implicit flows) and declassification (including conditional gradual release [9]) in a way that is both expressive and avoids scattering and tangling. Declassifications in the program should be guaranteed to follow the stated policy. Moreover, policy decisions about when, where, and how information may be declassified should not be scattered throughout the program and mixed together (tangled with) the program's code. The policy should be written separately from the code, which may make it possible for the policy to be reused. In particular, security auditing should be easy to accomplish even when a program changes, in the sense that the effort needed for auditing should be independent of the size of a program to which the policy is applied and the number of changes made to the program during maintenance.

Our approach to solving these problems is embodied in a new policy language, named *Evidently*. In *Evidently* policies are written separately from the program in an aspect-like fashion. The language can define security labels, models that connect the policy to the data locations in a program, and policies that express both normal flows and when declassifications may occur (following the work of Banerjee and Naumann [9]). Declassifications may depend on program states as identified in the model. The model in an *Evidently* policy is the only direct connection to the program, and can act to insulate the rest of the policy from program changes; thus if the program changes, at most the model will need to change.

We assume that the only observations that an attacker may make occur in the source language; that is we ignore covert channels and take the point of view that what matters about a program is its external observations. We adopt a termination insensitive semantics; that is, we do not consider termination or lack of termination to be observable.

The key contributions of this work include:

1. The design of a novel specification language for describing the information flow requirements of programs. In this paper we provide a description of our language as well demonstrate its use within several case studies.
2. An implementation of our language which describe and enforce information flow policies for programs written in the Java programming language. In addition to our implementation, we demonstrate how to enforce information flow by combining Aspect-oriented programming techniques with data tag tracking within the Java Virtual Machine.
3. A presentation of several case studies that detail the problems one can encounter when using an information flow specification language that is interwoven with the program and their solution in *Evidently*.

Comparison of Evidently with Existing Approaches

Although there are a number of other information flow policy languages such as JIF [70] and others [68, 87, 84, 75, 69, 14], few languages have focused on the issue of separating an information flow policy from an application's code. As such, the traditional approach has been to encode information flow policies directly in association with the code one wishes to apply an information flow policy to.

In its aim, Jeeves [99] is a recent proposal that has been designed to tackle the problem of separating information flow policy from code and is most similar to *Evidently* in intent.

Language Design: The design approach of Jeeves is not to create a single language that can describe information flow policies of multiple languages. Rather, the current Jeeves implementations are embedded into the languages they work with. For example, in the Python implementation of Jeeves, the programmer imports the Jeeves library, and annotates program methods with special `@jeeves` annotations and other Python functions that are defined in the Jeeves library. *Evidently* is a stand alone language that does not require the programmer to import libraries, or write additional code to enforce policies. In our formalization of *Evidently* we have separated language specific concerns (such as differences between statically typed and dynamically typed languages) from the core syntax so that *Evidently* may easily be adapted to other languages.

Policy Seperation: In Jeeves, to create a policy, one writes additional program code in the language that the original program is written in. For example, the following is an example of a Jeeves policy that introduces a faceted variable `value`, wherein the "high" view of the value is 42 and the "low" value is 20. This would be introduced into a Python program as follows:

```
x = JeevesLib.mkLabel()
JeevesLib.restrict(x, lambda ctxt : ctxt == 42)
value = JeevesLib.mkSensitive(x, 42, 20)
self.assertEquals(JeevesLib.concretize(value,value), 42)
```

In this model, the person implementing the security policy must write these statements into the program code and explicitly deal with the label types that Jeeves introduces.

In *Evidently*, the security policy is wholly contained within the source code of *Evidently* policy files. It is the responsibility of the *Evidently* compiler and runtime to locate the places in the program code where the correct instrumentation must be applied in addition to actually instrumenting it. In our current implementation, *Evidently* instruments Java byte code directly in a fashion similar to that found in Aspect-oriented programming. It should be noted however that the standard AspectJ language on its own is not enough to accomplish this since the current AspectJ langauge lacks the ability to describe locations within method bodies with enough precision for information flow tracking.

Implementation: Both *Evidently* and Jeeves offer implementations based on runtime checking of information flow. The current implementation of Jeeves, however, uses faceted execution [5, 6], a type of computation performed over a pair of executions, which allows it to track and detect implicit information flow. The current *Evidently* runtime leverages Phosphor [11], which can track indirect flow, but Phosphor is unable to track implicit flow. Part of the innovation of our work is that we have augmented Phosphor's capabilities to be able to handle implicit flow by implementing the *no sensitive upgrade* approach to handling implicit flow, i.e., we do now not allow upgrading of sensitive data locations when the current path condition depends on the value of a sensitive data location. We hope to improve this aspect our of implementation further in the future but do not consider it part of the design of the *Evidently* language itself.

Comparison with Aspect-Oriented Programming

In *Evidently*, the *flowpoints* of a model describe the data locations that can be used in the information flow policy. Flowpoints in turn utilize a syntax similar to that found in AspectJ for describing pointcuts. While the aspect-like syntax has been useful for describing flowpoints, *Evidently* does not contain the usual features found in aspect-oriented languages such as advice. As discussed in Section 5, we utilize AspectJ for describing method boundaries and performing information flow control. However, as noted by Yang [98], current implementations of aspect-oriented compilers lack the pointcut granularity needed to implement an information flow control system. Other authors have described fine-grained aspect languages [64, 80] that in concept could be useful if they had implementations available. However, while having sufficiently fine granularity is a necessary prerequisite for implementing a information flow control monitor, one would still have to implement many of the details built into a system like *Evidently* such as security labels, handling of implicit flow, and policy encoding.

Background on information flow Policies

Evidently has been designed to describe information flow policies. As described by an *information flow policy*, a perfectly secure system must have two properties: (1) no secure information is released to insecure program locations, and (2) no insecure program locations may influence the values held by secure program locations. These two properties are refereed to as "confidentiality" and "integrity" respectively. Formally, we say that a perfectly secure system has a property known as *noninterference*. The noninterference property is a security property of programs that places a constraint on the interaction of program locations. *Integrity* requires the flow of information in a program to be such that, if the program were to be executed multiple times with different insecure inputs, then in each execution the computation on the secure inputs would be performed identically. *Confidentiality* requires that for any set of set of runs wherein the insecure input values are held constant but the secure values are changed, there should be no difference in the insecure output values.

While a system with perfect confidentiality and integrity may sound ideal, such systems are generally not useful in real applications. For example, imagine a system that provides a login prompt to a user. If the user enters the wrong login information, the system must not allow the user access.

However, this action reveals something about what values the username and password *are not* and therefore leaks information. Such a system does not have the noninterference property.

Because the noninterference property is not practical in real systems, some amount of information release must be permitted. Systems that release information (and thus do not conform to noninterference) are said to *declassify* information. However, how much information should be released? What should be released? Under what conditions? The answer to these questions as well as several other important problems are specified by the *Security Policy*.

In *Evidently*, the security policy serves three important purposes. First, as was discussed earlier in this section, one way to reason about the information flow properties of a program is to consider each location in a program to be labeled with some value that denotes the intended security properties of that location. For example a program location that is very secure might be labeled H and a program location that is less secure might be labeled as L. Note that these labels may be more complex than the two-valued example given here. In the system described in this paper, a security label may take on many different values. For example, consider the set of labels $\{INTERNET, DATABASE, CAMERA\}$. Unlike the example with H and L it is not clear which label should be allowed to flow to where. Indeed, the decision to allow a flow is a decision that must be made on an application-by-application basis. Encoding these decisions is of paramount importance and thus it is the first responsibility of a security policy in *Evidently*. We explain how these relationships are encoded in Section 5.

Second, the noninterference property must be relaxed with some specified amount of declassification. However, this declassification must be carefully managed. The precise management of declassification is the second requirement of the *Evidently* language.

Finally, because the security policy must describe declassification policies for specific program locations and conditions, some degree of abstraction of the original program is necessary. Thus the final responsibility of *Evidently* is to provide an abstract environment for specifying program locations, security labels, and program states. This model is explained in detail in Section 5 and demonstrated in Section 5.

Declassification

One approach to declassification, called *delimited release*, has been identified by Sabelfeld and Myers [87]. It allows for special *declassify* annotations that explicitly release information where they are used in a program. It is then the responsibility of the security environment to ensure that *no more than* what is guarded by the annotation is released. In delimited release, the declassify annotation serves as an "escape hatch" that releases the information in the program where it is applied. One drawback to this approach is that the declassify annotation is arbitrarily powerful, that is, the locations in the program where it is applied are exempted from all checking. Because of how powerful they are, the use of these annotations is considered suspicious and each use requires manual review. Because of this manual review, the use of these annotations creates a potential source of errors during security auditing.

A more expressive approach to specifying declassification, called *conditioned gradual release*, has been identified by Banerjee et al. in their work on *flowspecs* [8]. Rather than using unchecked declassify annotations, conditioned gradual release requires that each location in the program where information is declassified form a valid *flowtriple*. Inspired by Hoare triples, flowtriples are of the form:

$$\{\varphi^{pre}\}\ C\ \{\varphi^{post}\} \tag{5.1}$$

In Equation (5.1), φ^{pre} and φ^{post} represent sets of pre/post state agreement predicates which are true only in the case that on two runs of a given program both runs agree on the values indicated in the predicates. As in Hoare logic, C is a program command. We will use the following program (used by both Sabelfeld and Myers as well as Banerjee et al.) to explain the meaning of Equation 5.1.

$$\textbf{if } (h \leq l) \textbf{ then } h := h - k;\ l := l + k; \tag{5.2}$$

In this example, the values l and k are considered to be low values and h is considered to be high (secret). In the flowspecs approach, to construct a flowtriple that states that the program is permitted to release information only about the relationship $h \leq l$, but no more information about h, (and that l and k are low values), one would write a flowtriple of the following form:

$$\{\mathbf{A}(h \leq l) \wedge \mathbf{A}(l) \wedge \mathbf{A}(k)\}\ C\ \{\mathbf{A}(l)\} \tag{5.3}$$

Although the flowtriples we have shown can precisely constrain the amount of information released, they are not yet policies. That is, they specify what is released, but they do not explain the conditions under which the information is released. To achieve this, Banerjee et al. augment the flowtriple with a state predicate that must be true in both runs in order for the release to occur. The combination of these two elements forms a flowspec policy.

Our aim in the design of *Evidently* is to support CGR. As explained in Section 5, the current implementation of *Evidently* handles both *indirect flow* and *implicit flow*. Our approach to declassification in the context of implicit flows is to use the *no sensitive upgrade* variant of declassification. The handling of indirect flow is provided by Phosphor, which is the system we have used to implement runtime data tagging for our implementation. Since Phosphor is unable to track implicit flows, the addition of *no sensitive upgrade* is a feature we have added in our implementation. *Evidently* uses the same style of declassification as CGR, i.e., rather than the delimited style of declassification, wherein declassifications are arbitrarily powerful relabelings, in *Evidently* policies may express the conditions for downgrading of sensitive information.

In *Evidently*, the declassification properties of a system are determined by a specified policy. The expressiveness of such a policy is measured in four dimensions: *what*, *where*, *who*, and *when*. Note that *Evidently* does not explicitly address the "who" dimension of declassification, which we leave to future work. To illustrate how *Evidently* specifies each of the other dimensions, we consider the hypothetical example of a medical records application below.

What. In a security policy, the "what" refers to the information being released. In our example, the "what" could be a record for a patient that is having their medical records transferred to another doctor. The "what" could also be more fine-grained—for example, a patient's diagnosis could be allowed to flow to their insurance company (but not, perhaps, their prior medical history). Note that this dimension of the policy need not have any minimal or maximal size; if viewed temporally, it is possible for it to be unbounded, for example, we may wish to release the results of a class of medical test and any further updates to the history of that particular test. In *Evidently*, the "what" is specified by Domain Models, which are explained in detail in Section 5.

Where. In the context of declassification, "where" can refer to policies that describe release to particular locations (variables or fields) in the code. One might like to describe particular variables or fields, perhaps those declared on particular lines of a program (code locality), or particular security levels of the information flow policy (level locality), or some combination of these. For example, a policy might specify that it is acceptable to declassify some a medical record from `doctors_office[i]` to `doctors_office[j]`.

Our approach takes the view that the where dimension should be described in terms of level locality, since it promotes a more centralized approach to policy construction, i.e., an analyst can use level locality to reason about a policy from the viewpoint of the policy file without having to know the exact positions in the code where information may be released. To put it another way, we take the view that the actual location is code where a piece of information is released is far less interesting than considering the underlying meaning of what the release might entail; knowing that the secret missile codes were released to the internet is far more interesting than knowing it happened on line number 2600. In *Evidently*, the “where” dimension is described via Levels, which are described in Section 5 which are in turn used to specify level locality in Policies, which are described in Section 5.

When. The when dimension of declassification is a temporal measure of the point in time (either abstract/state-based, or physical) beyond which it is acceptable to release information. The when dimension of declassification may refer to time in one of several ways. In their survey of declassification, Sabelfeld and Sands [86] identify three major classes of temporal releases: time-complexity based, probabilistic, and relative. In the time-complexity approach, declassification is permitted after a specific time. For example, a medical record may only be released to the hospital the night of the patient’s surgery. In the probabilistic model, a different view of noninterference is taken. Instead of noninterference being a binary property (the system has it or it doesn’t), the probabilistic model [86] describes systems in terms of the likelihood that a value may be released. Lastly, the relative approach permits declassification under an arbitrary condition. For example, our system may allow a patient’s records to be sent only after it has obtained a signature from the patient.

Evidently uses the relative approach, which may encode an arbitrary state (temporal or otherwise), so it is at least as powerful as the other two approaches (as it subsumes them). In *Evidently*, the when dimension can be encoded by combining properties (descriptions of states based on models) inside of a policy. Properties are described in detail in Section 5.

Motivating Examples

This section contains two examples in which a security policy has been applied to programs and demonstrates the problems that may arise if the security policy has been written into the source code of the program. We introduce these problems here and later, in Section 5, we revisit these examples and demonstrate how *Evidently* solves these problems. In Section 5, we introduce a example Android application which we will progressivly enhance with a *Evidently* policy through Section 5.

A Newspaper Changes its Subscription Model

A common occurrence in business software is that the software requirements change as the business changes. Consider an application that serves up news articles to website visitors. Since the site is new, it begins by offering up content to users for free and relies on ad revenue. However, as the site grows, the business decides that ad revenue alone is not enough the business and institutes a policy that each user will be given access to articles newer than 3 weeks only if they have read fewer than 10 articles that month. After that they will require a subscription to read articles from the archive.

The property in this example is a stateful one, namely, if a user has read 10 articles a month then they can access all articles, if not they can only access articles newer than 3 weeks. Stated as a conditional information property we might say that the site allows information to flow from the article database to the user only if the number of articles they have read so far is less than 11 or the article is older than 3 weeks old.

Depending on how this system has been written, updating the system to enforce this policy could be difficult. There could be many points at which articles are flowing to the database from other channels. Such modifications may be difficult and error prone to produce but are a good fit for *Evidently*'s design. We will discuss how this can be addressed in Section 5

A Growing Ecommerce Site

In addition to declassifying an entire sensitive value (for example, a credit card number), one may also wish to release only parts of sensitive values. In *Evidently* such a release is called a *projection*. Consider an online shopping cart wherein after a user submits their credit card number on a payment screen, we wish to redisplay the last 4 digits of their credit card number on a confirmation screen.

Now consider that over time our system grows. We now wish to allow users to log into the system to view previous orders, update payment information, and optionally save their credit card information in the system for faster checkouts. In each instance we may wish to display different projections of the sensitive values, i.e., the user's credit card number. In some cases we may wish to display the entire number, and in some cases we may wish to display only the last 4 digits for verification.

While it is inevitable that additional feature requirements will require new code to be written, although the security requirements of the application *have not* changed, for each new instance of displaying either the last 4 digits of the credit card or the entire credit card number our security analyst must select between two different types of declassification. In each instance a security analyst must read and understand new code and correctly apply the security policy to it. Such a process is error-prone and therefore undesirable in the context of security. What we would like instead is to be able to make more high level statements about the release of this information that will remain valid in the face of program modifications and be rejected if the program is coded in an insecure fashion.

Modeling Programs for Use in Security Policies

In *Evidently*, our aim is to provide a policy language which is centralized and not scattered throughout the program code. There are several advantages to this approach [29, 51]. First, scattering the contents of a security policy throughout a code base makes it harder to determine what the sum of all the parts of the security policy mean in relation to each other. This makes it difficult for an analyst to get a "bird's eye view" of the security policy of an application. Note that this problem cannot be addressed simply by collecting all the pieces of the policy throughout the codebase. It is common that these policy elements are tangled with code and therefore viewing the policy annotation without the code would make understanding the whole policy difficult for an analyst. Second scattering also makes the policy less reusable: if it is to be applied to a new program, then the new program must be annotated from scratch. Furthermore, if a security policy has been updated, it must be refactored throughout the codebase which may be difficult. Additionally, underlying code may be refactored, thus changing the semantics of the policy. Third, as will be shown in Section 5, *Evidently*'s approach to specifying policy makes policies more modular, e.g., policies may depend on and reuse other policies.

To achieve the effect of policy centralization, rather than scattering the code throughout the program code, we require the user to describe an abstraction of the program they wish to reason about which may then be used internally throughout the policy. In *Evidently* there are several such abstractions at a user's disposal: models, properties, projections (a special type of property), and levels. Ultimately, these 4 abstractions describe a policy, which is the ultimate abstraction in the language. In the following sections we describe each of these elements, starting with models, which are used to describe sets of memory locations of the program that a user is interested in. For brevity we omit the deals of more standard language issues such as the module/import system used in *Evidently* and instead demonstrate its features throughout this section.

Running Example: Decryption Key Release

Throughout this section of the paper we will be introducing the syntax of *Evidently*. To help motivate the discussion and the examples we show we will be referring to an example application; this is a mobile application for Android in which we have obtained an encrypted movie. The security policy is that we want to be able to view the movie, but *only after* a certain time (the release date). For simplicity, we express the release date as an Android `CountDownTimer` within the app. After the timer has elapsed we wish to make the video available to the user by releasing the decryption key and therefore the video. Note that this policy expresses both coarse and fine-grained rules. *Coarsely*, we wish that the movie may be stored in the database, that the user should be able to always see the current value of the timer but *finely* the user should only be able to view the video after a window of time has elapsed. This represents a conditional release of information. Furthermore, we exercise the projection aspects of our policy language by allowing the user to always know the *approximate* length of the movie (e.g., "about 90,000 bytes"), but not its exact length.

Models

The first *Evidently* policy abstraction we describe is models. The purpose of a model is to describe the sets of data locations in the program we wish write a policy about. A Model may describe a single data location or a collection of data locations. In addition to providing a method to identify data locations, Models may also define properties which may then be used in policy files. We provide a description of the syntax of Models in *Evidently* in Figure 5.1.

Flowpoints

So that we may describe program data locations in a way that is abstracted from the program text, the most fundamental aspect of Models is the method by which methods identify data locations. In *Evidently* this is done with the **flowpoints** keyword. This construct generally refers to a set of data locations in a program, but such a set may only contain one location if desired.

Returning to our running example, consider the task of reading an encrypted movie from the file system. To write a policy about this object, we are interested in all the data locations in the program where a file system object is read from a particular data location. To allow a user to be as specific or general as they need, provide a pointcut-like predicates and operators for describing sets of data locations. We provide a description of those predicates in Section 5.

In Listing 5.1, we define the flowpoint `movie`. In this listing we've indicated this particular location will be found in the package `examples.DecryptionKeyRelease`, (the **within** predicate) and that any receiver of the two specified variants of `getDecryptedVideo` should be counted as an instance of the movie location.

```
model Movie {

  flowpoints movie:File = {
    within(examples.DecryptionKeyRelease) &&
    (
     resultof(MCountdownActivity.getDecryptedVideo(String id))
     ||
     resultof(MCountdownActivity.getDecryptedVideo(File id))
    )
  }

}
```

Listing 5.1: Example flowpoint definition.

To shorten the flowpoint declaration, we support a wildcard syntax inspired by the pointcut specification syntax in AspectJ. For example, using the ellipsis parameter wildcard syntax, we could rewrite the previous example to the code found in Listing 5.2.

```
model Movie {


  flowpoints movie:File = {
    within(examples.DecryptionKeyRelease) &&
    (resultof(MCountdownActivity.getDecryptedVideo(...)))
  }


}
```

Listing 5.2: Using flowpoint signature wildcards.

The code in Listing 5.2 means that and indicates that an invocation with any parameters will match. A user may specify more specific arguments such as the types or actual concrete values (numbers, and strings).

Flowpoint Predicates and Predicate Operators

As described in the previous section, one of the fundemental building blocks of an *Evidently* model are flowpoints. Flowpoints in turn are are comprised of predicate expressions. We provide a description of each of these predicates in Table 5.1.

The predicates described in Table 5.1 may be combined to form predicate expressions that describe a set of data locations. They may be joined with the operators **&&** and **——**. Since predicates denote operations on sets of data locations, the meaning on the conjunction and disjunction operators is that **&&** produces the intersection of two sets of data locations and **——** produces their union.

Model ::= **model** *Id* { *MBElems* }

MBElems ::= *MBElem* | *MBElems MBElem*

MBElem ::= *Use* | *Flowpoints* | *Property*

Flowpoints ::= **flowpoints** *Id* **:** *ProgType*={ *EvExp* }
  | **flowpoints** *Id*={ *EvExp* }

EvExp ::= (*EvExp*)
  | *EvPredExt* (*MethDescs*)
  | *EvScopePred* (*Id*)
  | ! *EvExp* | *EvExp* && *EvExp* | *EvExp* || *EvExp*

EvPredExt ::= **execution** | **resultof** | **this** | **cflow**

EvScopePred ::= **within** | **typeof** | **named** | **field**

MethDescs ::= *MethDesc* | *MethDescs*, *MethDesc*

MethDesc ::= *IdOrStar* | *IdOrStar* (*MethFormalDescs*)
  | *TypeOrStar IdOrStar* (*MethFormalDescs*)

IdOrStar ::= *Id* | $\star$

TypeOrStar ::= *ProgType* | $\star$

MethFormalDescs ::= ... | *TypeOrStar**,

Property ::= **property** *Id* **:** *ProgType EqSpec* { *PropBody* }
  | **property** *Id EqSpec* { *PropBody* }
  | **property** *Id PFormals* **:** *ProgType EqSpec* { *PropBody* }
  | **property** *Id PFormals EqSpec* { *PropBody* }

EqSpec ::= = *Specification* | =

Specification ::= *Requires Assignable Ensures*
  | *Assignable Ensures*
  | *Requires Ensures*
  | *Ensures*

Requires ::= **requires** *Exp*;

Assignable ::= **assignable** *Exp*;

Ensures ::= **ensures** *Exp*;

PropBody ::= *Exp*

PFormals ::= (*Formals*)

Formals ::= *Formal* | *Formals*, *Formal*

Formal ::= *Id* **:** *ProgType* | *Id*

Figure 5.1: Syntax of Models in the Evidently policy language.

Table 5.1: Flowpoint predicates available in *Evidently*.

Predicate Name	Meaning
execution	The set of data locations within the execution of the supplied argument.
resultof	The set of data locations that are the result of the argument's execution.
this	The set of data locations where the **this** keyword produces the specified type.
cflow	The set of data locations within the control flow of the supplied argument.
within	The set of data locations contained within the supplied namespace.
typeof	The set of data locations that have the type of the argument.
named	The set of data locations where the program identifier matches the supplied identifier.
field	The the set of data locations where the object field name matches the supplied argument.

Properties

In Evidently properties have two purposes: First, properties denote attributes of a model that may be released by a policy. Second, since one may wish to only release a portion of a value, properties also allow users to define *projections* of model values. For an example of the first kind of property, consider the code in Listing 5.3.

```
model Movie {

   flowpoints movie:File = {
    within(examples.DecryptionKeyRelease) &&
    (resultof(MCountdownActivity.getDecryptedVideo(...)))
  }

   property movieLength:long = {
      movie.length();
   }
}
```

Listing 5.3: Example property definition.

In Listing 5.3, we use the previously defined flowpoint to describe the data locations containing the movie file. The property `movieLength` defined here defines a long value resulting from calling the `length()` method of any member of the `movie` flowpoint set.

Property Projections

Sometimes it is never acceptable to declassify a sensitive value. In our example, we wish to declassify a movie (the encryption key used to decode the movie), but only after a certain window of time elapses. While it is important to ensure that the movie is not released, we may wish to tell the user something about the underlying information. For example, we may decide it is safe to tell the user approximately how long a movie is in bytes. In Evidently this is achieved through the use of property projections.

In Evidently, since properties may contain expressions, a projection is a function that can be applied to a member of a flowpoint set in order to release a sanitized version of the underlying value. In Listing 5.4 we show an example projection that will compute the approximate video size of the encrypted video to the nearest 1000 bytes.

```
model Movie {
  // flowpoints and properties elided

  // round to nearest 1000 bytes
  property approximateVideoSize:long = {
    ((movieLength + 500) / 1000) * 1000;
  }
}
```

Listing 5.4: A projection of the movie size property.

In Listing 5.4 we define a projection on the model Movie that releases approximately how long the underlying movie to the nearest 1000 bytes to the user. Note that defining a projection (or property) alone is not enough to release information. In order for a projected value to be released it must also be specified in a policy that the projection of this information is allowed to be released. The mechanism for information release is described in Section 5.

Property Specifications

One of the problems with projections is that it is difficult to establish and maintain trust in the code that produces the trusted value. For example, one may design a routine to expose the last four digits of a credit card and use that method to write a property projection in a model. However, over time this method may change. Implicitly, be designating this method as a *sanitizer* we are raising it above scrutiny from the automated tools. This can be problematic is the code changes in a way that degrades the security of the projection. To help address this, properties may also have *specifications* attached to them. We use a lightweight pre- and post-condition style specification language similar to the JML specification language [57].

As an example, consider the code in Listing 5.5 that adds a specification to our previous projection. In Listing 5.5, we have added three **ensures** clauses to the specification of this property. The meaning of the three extra clauses is that in order for the value of this property to be considered a valid projection it must satisfy the specification before it is released. *Evidently* currently supports preconditions with the **requires** keyword and postconditions with the **ensures**.

```
model Movie {
  // flowpoints and properties elided
  property approximateVideoSize:long =
    ensures \result % 10 == 0;
    ensures \result % 100 == 0;
    ensures \result % 1000 == 0;
  {
    ((movieLength + 500) / 1000) * 1000;
  }
}
```

Listing 5.5: Using a specification in a property definition.

Levels

In Section 5 and we discussed the role of security labels in our approach. Throughout these sections we have been using the informal H and L to represent arbitrary secure and insecure levels.

In *Evidently*, the set of valid labels usable within security labels are defined via Levels. Once defined these labels are recognized within a policy. A level may be defined as a sink (a place locations may flow to), a source (a place values may have come from), and both which means that a label can function as a sink and a source. In Listing 5.6 we show the set of levels used for our sample application. Though the labels need not follow any specific convention, the labels we show below are derived from the ones used in Sparta [30].

```
levels Android {

    sink {
          DISPLAY
        , SPEAKER,
        , WRITE_CLIPBOARD
        , WRITE_EMAIL
        , WRITE_LOGS
    }

    source {
          ACCELEROMETER
        , BUNDLE
        , MEDIA
        , PHONE_NUMBER
        , RANDOM
        , READ_CLIPBOARD
        , READ_EMAIL
        , READ_TIME
        , REFLECTION
        , SENSOR
        , USER_INPUT
    }

    both {
          CAMERA_SETTINGS
        , CONTENT_PROVIDER
        , DATABASE
        , FILESYSTEM
        , INTENT
        , PARCEL
        , PROCESS_BUILDER
        , SECURE_HASH
        , SHARED_PREFERENCES
        , SYSTEM_PROPERTIES
    }
}
```

Listing 5.6: Example levels definition for an Android application.

Policies

To encode the security properties of an application a user must author a security policy. In *Evidently*, the Model, Property, Projection, and Level abstractions are combined and used to express the security properties of a system via a Policy. In this section we will be discussing the policy features of *Evidently* as well as showing examples of how to encode common security properties in *Evidently* policies.

In our policy language, *releases* are specified as in Listing 5.7. In Listing 5.7 one can see how this maps nicely onto our dimensions of declassification. The *What* dimension is described by the first element of the tuple following the `release` keyword. Valid values for what is to be released are any Model (which releases all properties in that model) or a specific property of a model. The *Where* dimension is expressed by the `From->To` syntax. `From` and `To` may be any valid level label or set of label labels. Lastly, the *When* dimension is expressed by the keywords `when` and `unless`.

```
release (What, From->To) = {
    when {
        // conditions
    }
    unless {
        // conditions
    }
}
```

Listing 5.7: An example of a policy release statement.

Levels	::=	**levels** *Id* { *LevelsSpec* }
LevelsSpec	::=	*LevelElem*
	|	*LevelsSpec LevelElem*
LevelElem	::=	*LevelKind* {LevBodyElems}
LevelKind	::=	**sink**
	|	**source**
	|	**both**
LevBodyElems	::=	*Id*
	|	*LevBodyElems*`,` *Id*

Figure 5.2: Syntax for defining Levels recognized by the policy.

```
using levels Android;

policy ReleaseVideoAfterReleaseDate  {

    use model Movie;
    use model Timer;

    release (Movie, DATABASE->DISPLAY) = {
        when {
            Timer.releaseDatePast
        }
        unless {
            // don't release the Bluray version
            Movie.movieLength > 2000000000;
        }
    }

}
```

Listing 5.8: A policy releasing a movie if the release date has passed (except for if it is the Bluray version).

Keeping with our example, we can express the policy for our movie release application with the policy in Listing 5.8. This policy states that we should release all of the properties contained within the Movie model, but only if the release date has passed and the version of the movie isn't the BluRay version (we assume the length of the file being larger indicates it is the Bluray version). This intent is specified by the `when` clause. The `unless` clause states that it should be allowed unless the license is revoked. Note we omit the definition of the other model used in this policy (Timer) for brevity.

Template Parameters

To make policies more flexible, Evidently supports parametric polymorphism (like Java's Generics). The type parameters are declared after the name of the policy and once defined may be used within any element of the release tuple. As an example, we show a polymorphic version of our video release policy in Listing 5.9.

```
using levels Android;

policy ReleaseVideoAfterReleaseDate<F,T>  {

    use model Movie;
    use model Timer;

    release (Movie, F->T) = {
        when {
            Timer.releaseDatePast
        }
        unless {
            // don't release the Bluray version
            Movie.movieLength > 2000000000;
        }
    }

}
```

Listing 5.9: Using parametric types in a policy definition.

In Listing 5.9, we declare F and T to be the parametric types and we use them to specify the locations a value may flow from and to. To fix concrete types to these parametric types, they must be bound by including this policy (possibly by including it through the root policy, which is explained in Section 5). We show an example of this, below:

```
use policy ReleaseVideoAfterReleaseDate<DATABASE,DISPLAY> as movieFromDBToDisplay;
```

In this example, the parameter F takes on the label DATABASE and the parameter T takes on the label DISPLAY.

Policy	::=	**policy** *Id* { *PoBElems* }
	$\mid$	**policy** *Id GParams* { *PoBElems* }
GParams	::=	⟨ *Ids* ⟩
Ids	::=	*Id* $\mid$ *Ids*, *Id*
PoBElems	::=	*PoBElem* $\mid$ *PoBElems PoBElem*
PoBElem	::=	*Use* $\mid$ *Release* $\mid$ *EnforceDirective*
Use	::=	*UsePolicy*
	$\mid$	*UseElem*
UsePolicy	::=	**use policy** *Id*;
	$\mid$	**use policy** *Id* **as** *Id*;
	$\mid$	**use policy** *Id GArgs*;
	$\mid$	**use policy** *Id GArgs* **as** *Id*;
UseElem	::=	**use model** *Id*;
	$\mid$	**use model** *Id* **as** *Id*;
GArgs	::=	⟨ *GenericArgs* ⟩
GenericArgs	::=	*GenericArg* $\mid$ *GenericArgs*, *GenericArg*
GenericArg	::=	*Id* $\mid$ {*Ids*} $\mid$ $\star$
Release	::=	**release** (*EvidentlyId*, *LevelFlow*) ={*RTimes*}
	$\mid$	**release** ($\star$, *LevelFlow*) ;
LevelFlow	::=	*GenericArg* −> *GenericArg*
RTimes	::=	*RTime* $\mid$ *RTimes RTime*
RTime	::=	**when** { Exp } $\mid$ **unless** { Exp }
EnforceDirective	::=	**enforce** *EnforceMode Id*;
EnforceMode	::=	**static** $\mid$ **runtime**

Figure 5.3: Syntax of an Evidently Policy.

The Root Policy

In *Evidently*, the information flow policy of an application is centralized. However, to support better modularity, the language elements such as models, lenses, projections, and policies may be split over multiple files. To bring all of these elements together, our system requires a *root policy* file. As a convention this file is called `policy.epl` and it serves as the entry point into the applications information flow policy. Within this file the user may import other policies, fix polymorphic types, and set the enforcement level of the various policies. We show an example of this file for our application in Listing 5.10.

```
using levels Android;

policy DecryptionKeyReleaseApp {

    use policy ReleaseVideoAfterReleaseDate<DATABASE,DISPLAY> as movieFromDBToDisplay;
    use policy AlwaysReportApproximateVideoSize;

    enforce static movieFromDBToDisplay;
    enforce runtime AlwaysReportApproximateVideoSize;

}
```

Listing 5.10: An example root policy definition.

In Listing 5.10, the root policy imports two policies. The first policy fixes the labels DATABASE and DISPLAY to the parametric types in the policy as show in Section 5. Lastly, we see in the final two lines of this policy file the `enforce` keyword. Note that simply importing a policy does not cause enforcement of the policy; the enforcement mode must be set. If the keyword `static` is specified, the `when`/`unless` portion of the policy will be checked statically. Otherwise these constructs are checked at runtime.

Defining the Default Security Lattice

By default, any level may flow to itself. However, in a realistic system, other flows may be permitted without qualification.

For example, in our movie app, we may wish to collect user input and send it over the network. We would like to specify that this should happen without restriction. In evidently this is achieved by creating a policy that specifies these rules. The general approach is to create a policy that always releases information according to the default security rules in an application. Take for example the following portion of a policy:

```
release (*, USER_INPUT->NETWORK);
release (*, FILESYSTEM->DATABASE);
```

The release statements above unconditionally release information from the user input to the network and from the filesystem to the database. Since no conditional or projections properties are specified, release is always allowed. A series of policy statements such as the ones above form what *Evidently* calls the default security lattice as described by Denning [26].

Implementation

To demonstrate the generality of *Evidently*, we developed an implementation of our language for the Java language. In this section we provide details on our implementation.

Prior to implementation, we formalzied the grammar of *Evidently* in the K Framework [82], a rewrite-based executable semantic framework. Since one of the goals of *Evidently* is to be language independent, we seperated the langauge into two modules. The first module defines the Java-specific language details needed for *Evidently* to express policies about Java programs. In our implementation we reduced the size of the language specific module to contain just two syntatic classes.

To demonstrate the enforcement features of *Evidently*, we created a version of *Evidently* that checks information flow at runtime. To do this, we created an compiler for *Evidently* .

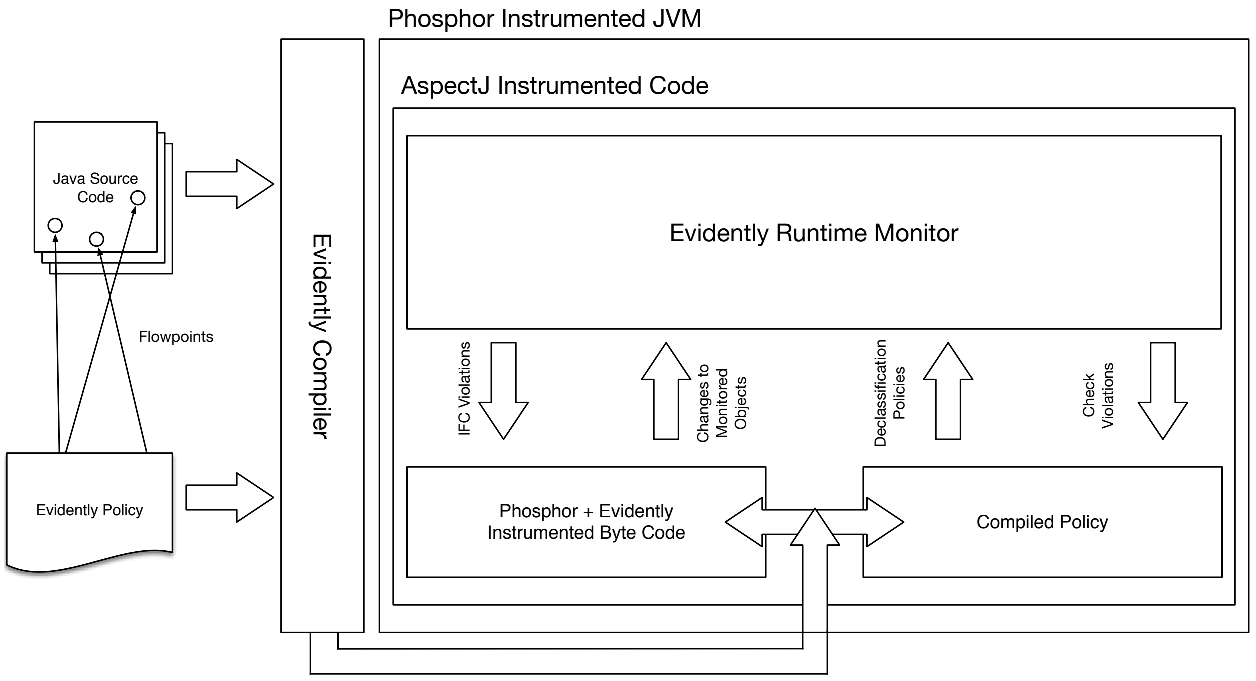

Phosphor Instrumented JVM
AspectJ Instrumented Code
Java Source Code
Evidently Compiler
Evidently Runtime Monitor
Flowpoints
IFC Violations
Changes to Monitored Objects
Declassification Policies
Check Violations
Evidently Policy
Phosphor + Evidently Instrumented Byte Code
Compiled Policy

Figure 5.4: An overview of the design of our Java implementation for runtime enforcement. In our approach, we leverage Phosphor to abstract the details of propagating our information flow labels. We handle enforcement by instrumenting the code with special calls to our runtime monitor and by attaching the AspectJ-based monitor to all the call and return points of methods.

We give an overview of our implementation in Figure 5.4. We implemented the *Evidently* compiler within the XText [32] language development environment. To handle runtime propagation of security labels we leveraged the Phosphor [11] tool for creating a specially instrumented indirect flow tracking variant of the Java Virtual Machine. Additionally, *Evidently* makes direct use of ASM (Java bytecode generation) in three ways. First, we use ASM for instrumenting the flowpoints within the target source code with special annotations designed for integration with Phosphor. Second, as a part of our implementation we also perform additional bytecode generation which allows us to perform implicit flow tracking by following the *no sensitive upgrade* variant of declassification in implicit flow contexts [3, 4, 5]. Lastly, since current implementations of aspect-oriented compilers do not allow joinpoints to be specified for operations like method level assignments of local variables (which is necessary for doing information flow control), we performed bytecode generation to gain access to this information. This feature enabled us to implement the predicates in Table 5.1. Finally, we used AspectJ [50, 54, 52] to listen to all of the callsites within the application and use those boundaries to check the validity of information flow either before invocation or after a value is returned from a method. Our version of *Evidently* is available as an Eclipse plugin.

Motivating Examples Revisited

In this section we revisit the examples introduced in Section 5 and discuss how *Evidently* helps to avoid problems.

A Newspaper Changes its Subscription Model

Recalling the issues from Section 5, this example poses the problem of a newspaper website that wants to change its subscription model from being free (allowing unlimited access to all articles) to only allowing access to fresh articles if the user has read less than 10 free articles in the most recent 4 week period. This is problematic because there may be many places in the application where articles are released; not only must the business logic of tracking how many articles have been read be implemented, this logic must be also scattered throughout the code at all such points to effectively enforce the policy.

In *Evidently* , such a policy could be written like the code in Listing 5.11.

In the Listing 5.11 we define two models and a policy (note that for simplicity we restrict ourself to the security channels **H** and **L**). In the **Article** model we define a flowpoint **art** which represents all of the data locations where an article has flowed out of the database. Because there may be may signatures (parameters) to this method we do not specify them and instead rely on *Evidently*'s wildcard matching. To get an abstraction of the age an article, we create the property **articleAgeInWeeks**, which is a projection of a given article that determines its age in weeks. In the **User** model we create a single flowpoint which gives us access to a user at the point in the program before they authenticate. According to the flowpoint it is found within the **ViewManager** during the execution of **getArticleForUser**. Finally, in our policy **TrialUser**, we express the contents of our policy. One potential weakness of this policy is our definition of **preAuthUser**. In this case we assume that the pre-authenticated user exists within the execution of **getArticleForUser** (we rely on the **typeof** function to locate the user object). However, if there are many such places where a user is retrieved prior to releasing an article, our system will generate a warning for those locations and policies can be fashioned. Because an appropriate default security lattice would not allow such a flow to occur, we do not have to worry about potentially insecure flows happening outside the conditions of our policy.

```
model Article {

    flowpoints art:Article = {
        resultof(ArticleCMS.getArticle(...))
    }

    property articleAgeInWeeks(a:Article):int = {
       (System.currentTimeMillis() - a.createdAt()/1000L)  / ((long)7*60*60*24)
    }
}

model User {

    flowpoints preAuthUser:User = {
        within(ViewManager) && execution(getArticleForUser(...))
        && typeof(User)
    }
}

policy TrialUser<H,L> {

    use model Article;
    use model User;

    release(Article.art, H->L){
        when {
            articleAgeInWeeks(Article.art) <= 3
            &&
            AccountManagementInterface.getAccount(User.preAuthUser).articleCount() < 10
        }
    }
}
```

Listing 5.11: The security policy for the newspaper application.

```
model Card {

    flowpoints paymentCard:PaymentCard = {
        typeof(PaymentCard)
    }

    property card:String = {
        paymentCard.toString()
    }

    property lastFour:String =
        ensures \result.length() == 4;
        ensures \result.equals(card.substring(card.length()-4));
    {
        Utils.getLast4OfCard(card);
    }
}

policy ReleaseLast4<H,L> {

    use model Card;

    release(Card.lastFour, H->L);
}
```

Listing 5.12: The security policy for the Ecommerce site.

A Growing Ecommerce Site

For our second example, we would like to express a security policy in such a way that it enforces that no public channels can see the entire credit card number of a customer; on those channels instead we should display the last 4 digits. There are two problems at work here. First, we must contend with the potential scattering of references to the credit card number throughout a growing codebase. Second, we must make sure that sanitized credit card number in fact conforms to our security requirements. The *Evidently* policy in Listing 5.12 encodes these concerns.

In Listing 5.12, we capture all data locations containing a sensitive card value using the flowpoint **paymentCard**. To simplify the writing of the specification of **lastFour**, we define a new property called **card** that extracts the string representation of the credit card number. We then define the projection **lastFour** to call out to the utility library to extract the last four digits of the card. The attached specification ensures that the string produced by the utility library produces a string that conforms with the specification. Namely, that it is exactly 4 characters long and the last 4 match the last 4 of the card number it was executed on.

Related Work

Evidently uses the theoretical framework developed in several other works. For the baseline information flow, *Evidently* uses the seminal work of Denning [26]. In terms of the declassification model used in *Evidently*, we utilize the theoretical framework for declassification described by Banerjee and Naumann in their work on conditional gradual release [8].

One of the early, Java-specific examples of a language for describing information flow was the work developed by Myers and Liskov [85, 68] in the form of JIF [70]. *Evidently* differs from these proposals in several important ways. First, rather than expressing a decentralized policy (as in the case of JIF), *Evidently* focuses on providing a centralized policy. Secondly, in JIF, labels are afixed to a program directly in the code and each label itself contains a policy. In *Evidently* we take the view that this scattering of policy and labels throughout the code makes it difficult to understand the security policy and makes software evolution and maintenance challenging.

Perhaps most similar in spirit to *Evidently*, is the work of Yang et al. on Jeeves [99]. We discuss the differences of *Evidently* and Jeeves at length in Section 5. A major difference between

Evidently and Jeeves is in which the way both policies are introduced into code and the use of faceted values in Jeeves. In Jeeves one has the ability to attach policies where sensitive values originate, but it is done not in Jeeves itself but in the domain specific language implementation. In *Evidently* this is accomplished via our flowpoints mechanism. Secondly, in Jeeves faceted values are explicitly introduced. *Evidently* does not need a mechanism for introducing its labeled values since it delegates this responsibility to the compiler implementation.

CHAPTER 6: CONCLUSIONS

In this dissertation I present three different lines of research, each designed to address problems that arise from the use of specification in modern software systems.

In Chapter 2, I introduced a new system called Spekl that is aimed at streamlining the process of specification authoring and usage. As I have shown, Spekl's design solves the three key subproblems of achieving wider adoption of program specification: tool installation / usage, specification consumption, and specification authoring and extension. Additionally, I showed how Spekl addresses the problem of adaptive reuse for specifications by providing tooling to support writing and reusing specifications. In addition to providing a description of the problems impacting specification reuse, in this dissertation I have provided a detailed discussion of the Spekl system from the perspective of tool authors, specification authors, and users of both tools and specifications.

In addition to being a proof of concept tool, Spekl is an effort to widen the adoption of specification, verification, and validation in general. The Spekl tools can be downloaded from http://www.spekl-project.org, and people wishing to contribute to the development of Spekl may join the project via http://www.github.com/Spekl.

In Chapter 3 I introduced Strongarm, an approach to inferring practical postconditions based on a novel algorithm for simplifying specifications. This algorithm (FAR) takes advantage of the inherent structure of logical formulas produced by applying predicate transformers. Strongarm also includes several other transformations that ensure that the output specification is practical. I evaluated the performance of Strongarm on 7 popular Java libraries and found that Strongarm was able to infer 75.7% of the specifications for these libraries. Additionally, I found that the size of the inferred specifications were also practical, as 95.0% of the inferred specifications were less than 1 page in length and 84.6% of the inferred specifications were less 1/4 of a page in length.

In addition to the work formalized in this dissertation, Strongarm will be released as a part of OpenJML, the community standard for verifying programs with the Java Modeling Language [56]. Strongarm's features are available either via the command line version of OpenJML or via the Eclipse plugin which is distributed by OpenJML.[1]

In Chapter 5, I introduced *Evidently*, a novel policy specification language capable of expressing centralized information flow policies separate from the source code of applications. I introduced the syntax of *Evidently*, and motivated its design through the detailed presentation of several examples. In these examples, I present policies that are not centralized, i.e., scattered and tangled throughout the code, and showed how the design of *Evidently* would help reduce the issues with the decentralized policy approaches. Lastly, I provided a discussion of *Evidently*'s implementation, which combines concepts from Aspect-oriented programming, information flow, and byte-code instrumentation.

[1]OpenJML is available at the following URL http://openjml.org.

LIST OF REFERENCES

[1] Farhana Aleen and Nathan Clark. Commutativity analysis for software parallelization: Letting program transformations see the big picture. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, XIV, pages 241–252, New York, NY, USA, 2009. ACM. URL: `http://doi.acm.org/10.1145/1508244.1508273`, `doi:10.1145/1508244.1508273`.

[2] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *Proceedings of the 29th ACM SIGPLAN SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 4–16. ACM, 2002. URL: `http://doi.acm.org/10.1145/503272.503275`, `doi:10.1145/503272.503275`.

[3] Thomas H. Austin and Cormac Flanagan. Efficient Purely-dynamic Information Flow Analysis. *SIGPLAN Not.*, 44(8):20–31, December 2009. `doi:10.1145/1667209.1667223`.

[4] Thomas H. Austin and Cormac Flanagan. Permissive Dynamic Information Flow Analysis. In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS '10, pages 3:1–3:12, New York, NY, USA, 2010. ACM. `doi:10.1145/1814217.1814220`.

[5] Thomas H. Austin and Cormac Flanagan. Multiple Facets for Dynamic Information Flow. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 165–178, New York, NY, USA, 2012. ACM. `doi:10.1145/2103656.2103677`.

[6] Thomas H. Austin, Tommy Schmitz, and Cormac Flanagan. Multiple facets for dynamic information flow with exceptions. *ACM Trans. Program. Lang. Syst.*, 39(3):10:1–10:56, May 2017. URL: `http://doi.acm.org/10.1145/3024086`, `doi:10.1145/3024086`.

[7] N. Ayewah, D. Hovemeyer, J.D. Morgenthaler, J. Penix, and William Pugh. Using Static Analysis to Find Bugs. *IEEE Software*, 25(5):22–29, September 2008. `doi:10.1109/MS.2008.130`.

[8] Anindya Banerjee, David a. Naumann, and Stan Rosenberg. Expressive Declassification Policies and Modular Static Enforcement. *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 339–353, May 2008. `doi:10.1109/SP.2008.20`.

[9] Anindya Banerjee and David A Naumann. A Relational Program Logic for Data Abstraction and Security. pages 1–21, 2012.

[10] Mike Barnett and K. Rustan M. Leino. Weakest-precondition of Unstructured Programs. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '05, pages 82–87, New York, NY, USA, 2005. ACM. `doi:10.1145/1108792.1108813`.

[11] Jonathan Bell and Gail Kaiser. Phosphor: Illuminating Dynamic Data Flow in Commodity Jvms. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 83–101, New York, NY, USA, 2014. ACM. `doi:10.1145/2660193.2660212`.

[12] J. C. Bicarregui, C. a. R. Hoare, and J. C. P. Woodcock. The verified software repository: A step towards the verifying compiler. 18(2):143–151. URL: `http://link.springer.com/article/10.1007/s00165-005-0079-4`, `doi:10.1007/s00165-005-0079-4`.

[13] Alex Borgida, John Mylopoulos, and Raymond Reiter. &ldquo;&hellip;and nothing else changes&rdquo;: The frame problem in procedure specifications. In *Proceedings of the 15th International Conference on Software Engineering*, ICSE '93, pages 303–314, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press. URL: `http://dl.acm.org/citation.cfm?id=257572.257636`.

[14] Niklas Broberg, Bart van Delft, and David Sands. Paragon for practical programming with information-flow control. In *APLAS*, volume 8301, pages 217–232. Springer, 2013.

[15] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, 2005.

[16] Raymond P.L. Buse and Westley R. Weimer. Automatic documentation inference for exceptions. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 273–282, New York, NY, USA, 2008. ACM. URL: `http://doi.acm.org/10.1145/1390630.1390664`, `doi:10.1145/1390630.1390664`.

[17] Kyle Carter, Adam Foltzer, Joe Hendrix, Brian Huffman, and Aaron Tomb. SAW: The Software Analysis Workbench. In *Proceedings of the 2013 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '13, pages 15–18, New York, NY, USA, 2013. ACM. URL: `http://doi.acm.org/10.1145/2527269.2527277`, `doi:10.1145/2527269.2527277`.

[18] Stephen Chong and Andrew C. Myers. Security policies for downgrading. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, pages 198–209, New York, NY, 2004. ACM. URL: `http://doi.acm.org/10.1145/1030083.1030110`, `doi:10.1145/1030083.1030110`.

[19] Patrick Cousot, Radhia Cousot, Manuel Fahndrich, and Francesco Logozzo. Automatic inference of necessary preconditions. In *in Proceedings of the 14th Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'13)*. Springer Verlag, January 2013. URL: `http://research.microsoft.com/apps/pubs/default.aspx?id=174239`.

[20] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. Precondition inference from intermittent assertions and application to contracts on collections. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'11, pages 150–168. Springer-Verlag, 2011. URL: `http://dl.acm.org/citation.cfm?id=1946284.1946296`.

[21] Duncan Coutts, Isaac Potoczny-Jones, and Don Stewart. Haskell: Batteries Included. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell*, Haskell '08, pages 125–126, New York, NY, USA, 2008. ACM. URL: `http://doi.acm.org/10.1145/1411286.1411303`, `doi:10.1145/1411286.1411303`.

[22] Dan Craigen. Formal Methods Adoption: Whats Working, Whats Not! In Dennis Dams, Rob Gerth, Stefan Leue, and Mieke Massink, editors, *Theoretical and Practical Aspects of SPIN Model Checking*, number 1680 in Lecture Notes in Computer Science, pages 77–91. Springer Berlin Heidelberg, July 1999. URL: `http://link.springer.com/chapter/10.1007/3-540-48234-2_6`.

[23] Valentin Dallmeier, Nikolai Knopp, Christoph Mallon, Sebastian Hack, and Andreas Zeller. Generating test cases for specification mining. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 85–96, New York, NY, USA, 2010. ACM. URL: `http://doi.acm.org/10.1145/1831708.1831719`, `doi:10.1145/1831708.1831719`.

[24] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. Lightweight defect localization for java. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, ECOOP'05, pages 528–550. Springer-Verlag, 2005. URL: `http://dx.doi.org/10.1007/11531142_23`, `doi:10.1007/11531142_23`.

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

[26] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976. URL: `http://doi.acm.org/10.1145/360051.360056`, `doi:10.1145/360051.360056`.

[27] Edsger W. Dijkstra. Structured programming. chapter Chapter I: Notes on Structured Programming, pages 1–82. Academic Press Ltd., London, UK, UK, 1972. URL: `http://dl.acm.org/citation.cfm?id=1243380.1243381`.

[28] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1997.

[29] Tzilla Elrad, Mehmet Aksit, Gregor Kiczales, Karl Lieberherr, and Harold Ossher. Discussing aspects of aop. *Communications of the ACM*, 44(10):33–38, October 2001. URL: `http://doi.acm.org/10.1145/383845.383854`.

[30] Michael D. Ernst, René Just, Suzanne Millstein, Werner Dietl, Stuart Pernsteiner, Franziska Roesner, Karl Koscher, Paulo Barros Barros, Ravi Bhoraskar, Seungyeop Han, Paul Vines, and Edward X. Wu. Collaborative Verification of Information Flow for a High-Assurance App Store. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1092–1104, New York, NY, USA, 2014. ACM. `doi:10.1145/2660267.2660343`.

[31] 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. *Sci. Comput. Program.*, 69(1-3):35–45, December 2007. doi:10.1016/j.scico.2007.01.015.

[32] Moritz Eysholdt and Heiko Behrens. Xtext: Implement Your Language Faster Than the Quick and Dirty Way. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '10, pages 307–309, New York, NY, USA, 2010. ACM. doi:10.1145/1869542.1869625.

[33] Cormac Flanagan and K. Rustan M. Leino. Houdini, an Annotation Assistant for ESC/-Java. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, FME '01, pages 500–517, London, UK, UK, 2001. Springer-Verlag.

[34] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 234–245, New York, NY, USA, 2002. ACM. doi:10.1145/512529.512558.

[35] R. W. Floyd. Assigning meanings to programs. *Proceedings Symposium on Applied Mathematics*, 19:19–31, 1967.

[36] Robert W. Floyd. Assigning Meanings to Programs. In Timothy R. Colburn, James H. Fetzer, and Terry L. Rankin, editors, *Program Verification*, number 14 in Studies in Cognitive Systems, pages 65–81. Springer Netherlands, 1993. doi:10.1007/978-94-011-1793-7_4.

[37] Gordon Fraser and Andreas Zeller. Generating parameterized unit tests. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 364–374, New York, NY, USA, 2011. ACM. URL: `http://doi.acm.org/10.1145/2001420.2001464`, `doi:10.1145/2001420.2001464`.

[38] Mark Gabel and Zhendong Su. Javert: Fully automatic mining of general temporal properties from dynamic traces. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 339–349. ACM, 2008. URL: `http://doi.acm.org/10.1145/1453101.1453150`, `doi:10.1145/1453101.1453150`.

[39] Mark Gabel and Zhendong Su. Testing mined specifications. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 4:1–4:11, New York, NY, USA, 2012. ACM. URL: `http://doi.acm.org/10.1145/2393596.2393598`, `doi:10.1145/2393596.2393598`.

[40] Mike Gordon and Hélène Collavizza. Forward with Hoare. In A. W. Roscoe, Cliff B. Jones, and Kenneth R. Wood, editors, *Reflections on the Work of C.A.R. Hoare*, pages 101–121. Springer London, 2010. `doi:10.1007/978-1-84882-912-1_5`.

[41] Stijn de Gouw, Jurriaan Rot, Frank S. de Boer, Richard Bubel, and Reiner Hhnle. OpenJDKs Java.utils.Collection.sort() Is Broken: The Good, the Bad and the Worst Case. In Daniel Kroening and Corina S. Psreanu, editors, *Computer Aided Verification*, pages 273–289. Springer International Publishing, July 2015. URL: `http://link.springer.com/chapter/10.1007/978-3-319-21690-4_16`.

[42] Radu Grigore, Julien Charles, Fintan Fairmichael, and Joseph Kiniry. Strongest postcondition of unstructured programs. In *Proceedings of the 11th International Workshop on Formal Techniques for Java-like Programs*, page 6. ACM, 2009.

[43] Natalie Gruska, Andrzej Wasylkowski, and Andreas Zeller. Learning from 6,000 projects: Lightweight cross-project anomaly detection. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 119–130. ACM, 2010. URL: `http://doi.acm.org/10.1145/1831708.1831723`, `doi:10.1145/1831708.1831723`.

[44] Neelam Gupta and Zachary V. Heidepriem. A new structural coverage criterion for dynamic detection of program invariants. 2003.

[45] Michael Harder, Jeff Mellen, and Michael D. Ernst. Improving Test Suites via Operational Abstraction. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 60–71, Washington, DC, USA, 2003. IEEE Computer Society.

[46] Rich Hickey. The Clojure Programming Language. In *Proceedings of the 2008 Symposium on Dynamic Languages*, DLS '08, pages 1:1–1:1, New York, NY, USA, 2008. ACM. `doi:10.1145/1408681.1408682`.

[47] C. a. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972. `doi:10.1007/BF00289507`.

[48] Alexey Ignatiev, Mikoláš Janota, and Joao Marques-Silva. Towards Efficient Optimization in Package Management Systems. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 745–755, New York, NY, USA, 2014. ACM. `doi:10.1145/2568225.2568306`.

[49] Isaac Jones. *The Haskell Cabal, a common architecture for building applications and libraries*. 2005.

[50] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In J. Lindskov Knudsen, editor, *ECOOP 2001*

— *Object-Oriented Programming 15th European Conference, Budapest Hungary*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer-Verlag, Berlin, June 2001. URL: `http://aspectj.org/documentation/papersAndSlides/ECOOP2001-Overview.pdf`.

[51] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP '97—Object-Oriented Programming 11th European Conference, Jyväskylä, Finland, Proceedings*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, New York, NY, June 1997.

[52] Ivan Kiselev. *Aspect-Oriented Programming with AspectJ*. Sams Publishing, Indianapolis, 2003.

[53] Elisavet Kozyri, Owen Arden, Andrew C. Myers, and Fred B. Schneider. JRIF: Reactive Information Flow Control for Java. February 2016.

[54] Ramanivas Laddad. *AspectJ in Action*. Manning Publications Co., Grennwich, Conn., 2003.

[55] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, May 2006. `doi:10.1145/1127878.1127884`.

[56] Gary T. Leavens and Yoonsik Cheon. Design by contract with JML. Draft, available from jmlspecs.org., 2005. URL: `http://www.jmlspecs.org/jmldbc.pdf`.

[57] Gary T. Leavens and Yoonsik Cheon. Design by contract with JML. Draft, available from jmlspecs.org., 2005. URL: `http://www.jmlspecs.org/jmldbc.pdf`.

[58] Gary T. Leavens and Curtis Clifton. Lessons from the JML project. In Bertrand Meyer and Jim Woodcock, editors, *Verified Software: Theories, Tools, Experiments, Zurich, Switzer-*

land, volume 4171 of *Lecture Notes in Computer Science*, pages 134–143. Springer-Verlag, 2008.

[59] K. Rustan M. Leino. *This Is Boogie 2*. 2008.

[60] Hui Liu, Zhiyi Ma, Lu Zhang, and Weizhong Shao. Detecting duplications in sequence diagrams based on suffix trees. In *APSEC '06: Proceedings of the XIII Asia Pacific Software Engineering Conference*, pages 269–276. IEEE CS, 2006.

[61] Benjamin Livshits and Thomas Zimmermann. Dynamine: Finding common error patterns by mining software revision histories. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 296–305, New York, NY, USA, 2005. ACM. URL: `http://doi.acm.org/10.1145/1081706.1081754`, `doi:10.1145/1081706.1081754`.

[62] David Lo and Shahar Maoz. Mining hierarchical scenario-based specifications. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 359–370. IEEE Computer Society, 2009. URL: `http://dx.doi.org/10.1109/ASE.2009.19`, `doi:10.1109/ASE.2009.19`.

[63] Luqi and Joseph A. Goguen. Formal methods: Promises and problems. *IEEE Softw.*, 14(1):73–85, January 1997. URL: `http://dx.doi.org/10.1109/52.566430`, `doi:10.1109/52.566430`.

[64] Hidehiko Masuhara, Yusuke Endoh, and Akinori Yonezawa. A Fine-grained Join Point Model for More Reusable Aspects. In *Proceedings of the 4th Asian Conference on Programming Languages and Systems*, APLAS'06, pages 131–147, Berlin, Heidelberg, 2006. Springer-Verlag. `doi:10.1007/11924661_8`.

[65] T. Mens. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, May 2002. doi:10.1109/TSE.2002.1000449.

[66] Amir Michail. Data mining library reuse patterns using generalized association rules. In *Proceedings of the 22nd International Conference on Software Engineering*, ICSE'00, pages 167–176. ACM, 2000. URL: http://doi.acm.org/10.1145/337180.337200, doi:10.1145/337180.337200.

[67] Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine. Bayesian Specification Learning for Finding API Usage Errors. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 151–162. ACM. URL: http://doi.acm.org/10.1145/3106237.3106284, doi:10.1145/3106237.3106284.

[68] AC Myers and Barbara Liskov. A decentralized model for information flow control. *ACM SIGOPS Operating Systems Review*, (October), 1997.

[69] Andrew C. Myers. JFlow: Practical Mostly-static Information Flow Control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 228–241, New York, NY, USA, 1999. ACM. doi:10.1145/292540.292561.

[70] Andrew C. Myers and Barbara Liskov. Protecting Privacy Using the Decentralized Label Model. *ACM Trans. Softw. Eng. Methodol.*, 9(4):410–442, October 2000. doi:10.1145/363516.363526.

[71] Anh Cuong Nguyen and Siau-Cheng Khoo. Extracting significant specifications from mining through mutation testing. In *Proceedings of the 13th International Conference on Formal Methods and Software Engineering*, ICFEM'11, pages 472–488, Berlin, Hei-

delberg, 2011. Springer-Verlag. URL: `http://dl.acm.org/citation.cfm?id=2075089.2075130`.

[72] Hoan Anh Nguyen, Robert Dyer, Tien N. Nguyen, and Hridesh Rajan. Mining Preconditions of APIs in Large-scale Code Corpus. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 166–177, New York, NY, USA, 2014. ACM. `doi:10.1145/2635868.2635924`.

[73] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. Graph-based mining of multiple object usage patterns. In *Proceedings of the Symposium on Foundations of Software Engineering*, ESEC/FSE '09, pages 383–392. ACM, 2009. URL: `http://doi.acm.org/10.1145/1595696.1595767`, `doi:10.1145/1595696.1595767`.

[74] Jeremy W. Nimmer and Michael D. Ernst. Invariant Inference for Static Checking: An Empirical Evaluation. *SIGSOFT Softw. Eng. Notes*, 27(6):11–20, November 2002. `doi:10.1145/605466.605469`.

[75] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for java. In *Compiler Construction: 12'th International Conference, CC 2003*, volume 2622, pages 138–152, New York, NY, April 2003. Springer-Verlag.

[76] Michael Pradel and Thomas R. Gross. Automatic generation of object usage specifications from large method traces. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 371–382. IEEE Computer Society, 2009. URL: `http://dx.doi.org/10.1109/ASE.2009.60`, `doi:10.1109/ASE.2009.60`.

[77] Karthik Ram. Git can facilitate greater reproducibility and increased transparency in science. *Source Code for Biology and Medicine*, 8(1):7, February 2013. URL: `http://www.scfbm.org/content/8/1/7/abstract`, `doi:10.1186/1751-0473-8-7`.

[78] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. Static specification inference using predicate mining. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 123–134. ACM, 2007. URL: `http://doi.acm.org/10.1145/1250734.1250749`, `doi:10.1145/1250734.1250749`.

[79] Manos Renieris, Sébastien Chan-Tin, and Steven P. Reiss. Elided conditionals. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '04, pages 52–57, New York, NY, USA, 2004. ACM. URL: `http://doi.acm.org/10.1145/996821.996839`, `doi:10.1145/996821.996839`.

[80] Tobias Rho, Günter Kniesel, and Malte Appeltauer. Fine-grained generic aspects. In *Workshop on Reflection, AOP and Meta-Data for Software Evolution*, pages 29–35, 2006.

[81] Martin C. Rinard and Pedro C. Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Trans. Program. Lang. Syst.*, 19(6):942–991, November 1997. URL: `http://doi.acm.org/10.1145/267959.269969`, `doi:10.1145/267959.269969`.

[82] Grigore Roşu and Traian Florin Şerbănuţă. An Overview of the K Semantic Framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010. `doi:10.1016/j.jlap.2010.03.012`.

[83] Joseph R. Ruthruff, Sebastian Elbaum, and Gregg Rothermel. Experimental program analysis: A new program analysis paradigm. In *Proceedings of the 2006 International*

Symposium on Software Testing and Analysis, ISSTA '06, pages 49–60, New York, NY, USA, 2006. ACM. URL: `http://doi.acm.org/10.1145/1146238.1146245`, `doi:10.1145/1146238.1146245`.

[84] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003. `doi:10.1109/JSAC.2002.806121`.

[85] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Jan 2003. `doi:10.1109/JSAC.2002.806121`.

[86] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Computer Security Foundations, 2005. CSFW-18 2005. 18th IEEE Workshop*, pages 255–269, June 2005. `doi:10.1109/CSFW.2005.15`.

[87] Andrei Sabelfeld and Andrew C. Myers. A Model for Delimited Information Release. In *In Proc. International Symp. on Software Security (ISSS'03), Volume 3233 of LNCS*, pages 174–191. Springer-Verlag, 2004.

[88] Calvin Smith, Gabriel Ferns, and Aws Albarghouthi. Discovering Relational Specifications. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 616–626. ACM. URL: `http://doi.acm.org/10.1145/3106237.3106279`, `doi:10.1145/3106237.3106279`.

[89] Luka Stanisic, Arnaud Legrand, and Vincent Danjean. An Effective Git And Org-Mode Based Workflow For Reproducible Research. *SIGOPS Oper. Syst. Rev.*, 49(1):61–70, January 2015. URL: `http://doi.acm.org/10.1145/2723872.2723881`, `doi:10.1145/2723872.2723881`.

[90] Alexandru D. Sălcianu. jpaul – Java Program Analysis Utilities Library. Available from http://jpaul.sourceforge.net, 2005.

[91] Chris Tucker, David Shuffelton, Ranjit Jhala, and Sorin Lerner. OPIUM: Optimal Package Install/Uninstall Manager. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 178–188, Washington, DC, USA, 2007. IEEE Computer Society. URL: `http://dx.doi.org/10.1109/ICSE.2007.59`, `doi:10.1109/ICSE.2007.59`.

[92] Jrme Vouillon and Roberto Di Cosmo. Broken Sets in Software Repository Evolution. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 412–421, Piscataway, NJ, USA, 2013. IEEE Press. URL: `http://dl.acm.org/citation.cfm?id=2486788.2486843`.

[93] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. Detecting object usage anomalies. In *Proceedings of the Symposium on Foundations of Software Engineering*, ESEC-FSE '07, pages 35–44. ACM, 2007. URL: `http://doi.acm.org/10.1145/1287624.1287632`, `doi:10.1145/1287624.1287632`.

[94] Yi Wei, Carlo A. Furia, Nikolay Kazmin, and Bertrand Meyer. Inferring better contracts. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 191–200. ACM, 2011. URL: `http://doi.acm.org/10.1145/1985793.1985820`, `doi:10.1145/1985793.1985820`.

[95] Chadd C. Williams and Jeffrey K. Hollingsworth. Automatic mining of source code repositories to improve bug finding techniques. *IEEE Trans. Softw. Eng.*, 31(6):466–480, 2005.

[96] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal Methods: Practice and Experience. *ACM Comput. Surv.*, 41(4):19:1–19:36, October 2009. `doi:10.1145/1592434.1592436`.

[97] Tao Xie and David Notkin. Mutually enhancing test generation and specification inference. In *Formal Approaches to Software Testing: Third International Workshop on Formal Approaches to Testing of Software*, FATES '03, pages 60–69, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. URL: `http://dx.doi.org/10.1007/978-3-540-24617-6_5`, `doi:10.1007/978-3-540-24617-6_5`.

[98] Jean Yang and Institute of Technology Massachusetts. *Preventing Information Leaks with Policy-Agnostic Programming*. PhD thesis, 2015. OCLC: 940778536.

[99] Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. A language for automatically enforcing privacy policies. *POPL*, pages 85–96, 2012.

[100] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. Perracotta: Mining temporal api rules from imperfect traces. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 282–291. ACM, 2006. URL: `http://doi.acm.org/10.1145/1134285.1134325`, `doi:10.1145/1134285.1134325`.

[101] Apt - Debian Wiki. `https://wiki.debian.org/Apt`. URL: `https://wiki.debian.org/Apt`.

[102] Yum (Yellowdog Updater, Modified). `https://www.phy.duke.edu/~rgb/General/yum_HOWTO/yum_HOWTO/`. URL: `https://www.phy.duke.edu/~rgb/General/yum_HOWTO/yum_HOWTO/`.