EXTENDING DYNAMIC INVARIANT DETECTION WITH EXPLICIT ABSTRACTION

by

DANIEL BRIAN KEITH

A DISSERTATION

Presented to the Department of Computer and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

March 2012

DISSERTATION APPROVAL PAGE

Student: Daniel Brian Keith

Title: Extending Dynamic Invariant Detection with Explicit Abstraction

This dissertation has been accepted and approved in partial fulfillment of the requirements for the Doctor of Philosophy degree in the Department of Computer and Information Science by:

Michal Young	Chair
Zena Ariola	Member
Christopher Wilson	Member
Edward Vogel	Outside Member

and

Kimberly Andrews Espy	Vice President for Research & Innovation/ Dean of the Graduate School

Original approval signatures are on file with the University of Oregon Graduate School.

Degree awarded March 2012

DISSERTATION ABSTRACT

Daniel Brian Keith

Doctor of Philosophy

Department of Computer and Information Science

March 2012

Title: Extending Dynamic Invariant Detection with Explicit Abstraction

Dynamic invariant detection is a software analysis technique that uses traces of function entry and exit from executing programs and infers partial specifications that characterize the observed behavior. The specifications are reported as logical precondition and postcondition expressions (invariants) that relate arguments, instance variables, and results. Detectors typically generate large collections of invariants, among which most are true but few are interesting or useful. Refining this flood of invariants into a useful subset often requires manual tuning through configuration options and modification of the program under analysis.

Our research asks whether we can improve dynamic invariant detection by enabling explicit abstractions to be declared and applied to a program under analysis and whether this is practical; this dissertation shows that it is indeed practical and useful. Given a concrete program we can synthesize a model program composed of functions and modules that are abstractions of selected concrete modules. When we execute the model program in parallel with its underlying concrete program and apply dynamic invariant detection, we obtain abstracted invariants that can reveal the behavior of the underlying concrete program.

We developed the Alembic system to support and experiment with the above technique, enabling a practical method for steering the invariant detection process and shaping the analysis to produce more refined results than obtainable via traditional means. Alembic provides a simple language for defining abstractions and managing detection experiments; the system generates the necessary instrumentation, representation classes, and functions, freeing the analyst to focus on the expression of abstractions and detection experiments.

Alembic currently leverages the invariant detection capability of Daikon, a powerful first-generation detector, to analyze synthetic traces on abstractions. However, the principles we demonstrate apply to any detector and language that observes function entry and exit. We present some applications of this technique to example problems and then evaluate Alembic on production code such as the Guava class library. Our research suggests new uses for existing detectors and enables the design and evaluation of features to inform the next generation of dynamic invariant detection systems.

This dissertation includes previously unpublished co-authored material.

CURRICULUM VITAE

NAME OF AUTHOR: Daniel Brian Keith

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon, Eugene
Northwestern University, Evanston, Illinois

DEGREES AWARDED:

Doctor of Philosophy, Computer and Information Science, 2012, University of Oregon
Master of Science, Computer and Information Science, 2008, University of Oregon
Bachelor of Science, Computer Science, 2004, Northwestern University

AREAS OF SPECIAL INTEREST:

Programming Language Semantics
Logic and Proof Theory

PROFESSIONAL EXPERIENCE:

Research Assistant, Department of Computer and Information Science, University of Oregon, Eugene, 2010-2012

Teaching Assistant, Department of Computer and Information Science, University of Oregon, Eugene, 2005-2010

GRANTS, AWARDS AND HONORS:

Graduate Teaching Fellowship, Department of Computer and Information Science, 2007 to present

PUBLICATIONS:

Keith, D., Young, M., and Smaragdakis, Y. 2012. Extending dynamic invariant detection with explicit abstraction. Submitted to ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE) 2012.

Ariola, Z., Herbelin, H., Herman, D., and Keith, D. 2011. A robust implementation of delimited control. In TPDC 2011: Theory and Practice of Delimited Continuations 2011.

Keith, D., Hoge, C., Frank, R., and Malony, A. 2006. Parallel ICA Methods for EEG Neuroimaging. In IPDPS 2006: Proceedings of the 2006 IEEE International Parallel & Distributed Processing Symposium.

ACKNOWLEDGEMENTS

I am grateful to the Department of Computer and Information Science for providing support and a positive environment for my education and research, and I humbly acknowledge the support of the National Science Foundation for funding the research described herein. I am indebted to my dissertation committee, who accommodated a highly constrained thesis defense schedule to allow me to finish this Winter. Thank you so much, Chris, Ed, Zena and Michal.

During my time in the graduate program, I had the unusual fortune to work in three entirely different areas in computer science, with three excellent advisors. My first advisor, Zena Ariola, fulfilled my need to understand the scope, power and intricacies of formal language, logic and proof. My second advisor, Matthew Sottile, shared my love of programming and enabled me to build interesting and practical implementations of abstract ideas, without leaving the ethereal level of pure functional programming languages. I am especially grateful to my third and final advisor, Michal Young, who helped me find a research path where I could contribute my perspective and ultimately complete this work.

Along the way, I benefited from the tutelage of David Spivak and Patrick Schultz, who helped me understand the ubiquity and applicability of category theory and gave me a new mental tool with which to see all worlds. They were incredibly patient with my questions and tolerant of the mismatch in language and concepts between my computer science and their mathematical viewpoints.

The folks in the front office, Star Holmberg, Cheri Smith, and Jan Saunders, were invaluable in helping me navigate the necessary administrative waters, and a pleasure to talk to as well. I appreciate their help and patience and especially Star putting up with my unerring ability to ask her a question that she had already answered in a previous email.

I also want to thank the friends who supported me, including Doug Yook and Chris Hoge, who helped immensely by listening to my fretting and griping long enough for me to get back to work, and Don Macnaughtan, who encouraged me and set a great example with the completion of his book.

Finally and most importantly, my progress and success as a graduate student was only possible with the patience and support of my amazing wife, Melanie, and the tolerance of my daughters, Indica and Nikola. I would not and could not have done this alone, and am fortunate to have such a wonderful family. I hope that I can make up for time lost.

For my Mom and Dad.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Table | Page

LIST OF LISTINGS

Listing Page

Listing Page

Listing Page

Listing Page

CHAPTER I

INTRODUCTION

A dynamic invariant detector characterizes the observed behavior of a program under analysis as a set of logical clauses called *invariants*. Most detectors report a large amount of useless invariants along with a small amount of genuinely relevant invariants, placing a burden on the analyst to discern which invariants are useful and relevant. We developed a technique, *explicit abstraction*, for enhancing the quality of reported invariants by allowing the shaping and focusing of the detection process. We developed a domain-specific language, Alembic, to facilitate this abstraction technique and to make using dynamic invariant detection more fruitful.

We begin this chapter with a brief summary of the software engineering concepts of specification and analysis, which leads to the introduction of the dynamic invariant detection technique, and some of its applications. We point out some of the well-known problems associated with dynamic invariant detection, and use these to informally introduce our idea of explicit abstraction as a way to shape and refine the invariant detection process to produce higher-level descriptions of modules under analysis. We sketch this technique by applying it to the MinMaxPriorityQueue container class from the Guava Collections Library [Google, 2011]. Chapter III expands on this technique and how it is facilitated by the Alembic language[1], which we evaluate more fully in Chapter IV by applying Alembic to other classes in the Guava library.

1.1. Specifications and Contracts

A modern medium or large-scale software system is quite complex, usually comprising reusable third-party libraries, application-specific libraries, and driver programs to execute the ensemble according to the needs of the system. Partitioned, compositional design of such software is essential to the understanding, development, and maintenance of these systems. The use of well-defined interfaces and datatypes for limiting and clarifying the interactions of components helps manage the complexity of these systems.

Software interfaces that isolate the implementation of a component from the services it presents to its clients are common and expected in developing modern software. Less

[1]The name *Alembic* derives from an ancient distillation apparatus that enables separation and concentration of essential chemicals and scents from a dilute mixture; alembics have been used for medicine, alcohol and perfume production. The name was chosen to reflect the way that abstraction allows important information and invariants to be refined or distilled from the dilute sea of true, but, irrelevant invariants.

common in practice are interfaces augmented with the use of structural invariants and assume-guarantee specifications (software contracts) to increase the clarity, robustness and verifiability of software libraries. These rich specifications can serve as abstractions that can be used to aid program understanding, or to clarify the requirements for an implementor. These same specifications can also be used to mechanically generate or verify the coverage of unit tests, and as a basis for further analysis and verification tasks [Ernst et al., 2007].

In the next chapter, we will look at specifications in more depth, including a discussion of specification languages, enforcement and mining. For our purposes in this introduction, we confine ourselves to the following informal notion of module specification:

> A *specification* of a module is a description of the member functions of that module that constrains possible implementations, as well as possible sub-specifications. A specification describes *what* a module does, but not *how* it does it. We use the term *module* to refer to both object-oriented classes as well as package-oriented collections of functions and state.

We structure this specification by associating a set of logical *precondition* expressions with each function's entry, and a set of logical *postcondition* expressions with each function's return. These precondition and postcondition expressions of a function, collectively referred to as the function's *invariants*, constrain the possible implementations of the function, as well as constraining legitimate sub-specifications of the function.

A precondition of a function is a logical expression composed of a subset of the variables visible at function entry, as well as expressions built from these variables. These variables include the function's formal arguments, the keyword this, and any state or member variables visible from within the function. For example, the precondition for a square root function, sqrt, over the real numbers would likely include a clause requiring the argument n to be a non-negative number:

    n >= 0

If this specification were included along with an implementation of sqrt, then we would interpret it as saying that the function is only defined for non-negative integers, and has unspecified behavior otherwise.

A postcondition of a function is similar to a precondition, except that it may refer to both the original and final values of state variables and arguments, as well as to the function result. In this way, a postcondition describes some aspect of how a function behaves. For example, the postcondition for the square root function, sqrt, should include a clause that relates the function's result, return, to its argument, n:

return * return == n

A convenient way to represent the specification for a module is as a table. For example, we can imagine a ToyMath module containing the above sqrt function, as well as a sqr and floor function; the table in Table 1.1 conveniently displays the preconditions and postconditions.

Table 1.1. A tabular specification of the ToyMath module that displays the preconditions and postconditions for each of the three methods in the module.

Precondition	**Method**	**Postcondition**
none	float sqr(float n)	return == n**2
n >= 0	float sqrt(float n)	return**2 == n
none	int floor(float n)	n >= return n < return + 1

Specifications provide *behavioral* information that supplements the *type* information supplied by the function's signature. Alternatively, specifications can be seen as a way to more clearly specify the domain of a function's types than can be specified with syntactic data types.

Specifications consisting of precondition/postcondition clauses such as the above are often referred to as *contracts*, where the alternative names *assume/guarantee* or *require/ ensure* are used to identify precondition or postcondition clauses, respectively [Meyer, 1992]. The term contract refers to the idea that a specification promises to ensure a postcondition after a function executes, provided that the caller satisfied the precondition before execution.

Our ToyMath module has no state variables; however, most object-oriented classes have member variables that record the current state or contents of the object. Invariants over these classes can include member variables as well as arguments and results. In the case of stateful modules or classes, there is a notion called the *module invariant* or *class invariant* that refers to a predicate that must hold when the module or class is in an observable state (i.e., before and after any public method) [Liskov and Wing, 1994]. Detectors like Daikon (and layered extensions such as Alembic) infer these invariants by finding clauses that satisfy both the precondition and postcondition of all of the observed methods of a class.

1.2. Static and Dynamic Analysis

Software analysis techniques infer analysis-specific properties like resource metrics, safety and liveness, and characterizations of variable domains and function behavior. These can be crudely partitioned into *static* and *dynamic* techniques. Static analysis examines the text of implementations (in source or compiled form), whereas dynamic analysis observes executing implementations and the transient values and properties associated with this execution.

Given a module and a specification that purports to describe the module, there are static analysis techniques that can analyze the source code and verify that the implementation conforms to the specification (static verification). We can also use a specification to mechanically generate unit tests that attempt to exercise the implementation (test generation), or to dynamically verify that the implementation conforms to the specification (specification checking). Finally, we can use a specification to aid in understanding a module's behavior and intent; this can guide the development of a satisfying implementation, clarify and communicate assumptions about a module's behavior, or make it easier for a client to use the module.

In the case where we have no specification for a module other than its implementation and signature, we can apply software analysis techniques to discover these specifications (specification mining). Static analysis techniques examine implementations and derive specifications from these. They can be used at compile-time during the development of an implementation to ensure that the specification is being honored. In many cases, static analysis can provide an assurance that the analysis is comprehensive, and that all possible paths or situations have been analyzed. See Figure 1.1 for a schematic of static analysis, where the primary input is source code for a target module, and the output is a report of discovered properties. These properties can include a specification, or an observation that no exceptions are thrown, for example.

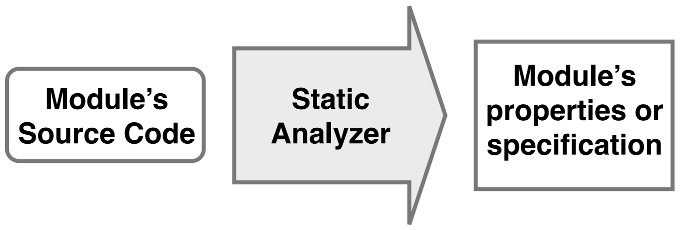


Figure 1.1. Static Analysis of the source code infers properties, optimizations and specifications without actually executing the code.

In contrast to static techniques, dynamic techniques are suitable for observing properties that are not easily computable from the text of an implementation. These properties include resource metrics such as processor time and space, as well as properties derived from linked libraries that cannot be analyzed statically.

A particular dynamic analysis technique known as *dynamic invariant detection* analyzes modules that are executing in some test harness or in a live deployment, and infers specifications from the observations of function entry and exit (Figure 1.2).

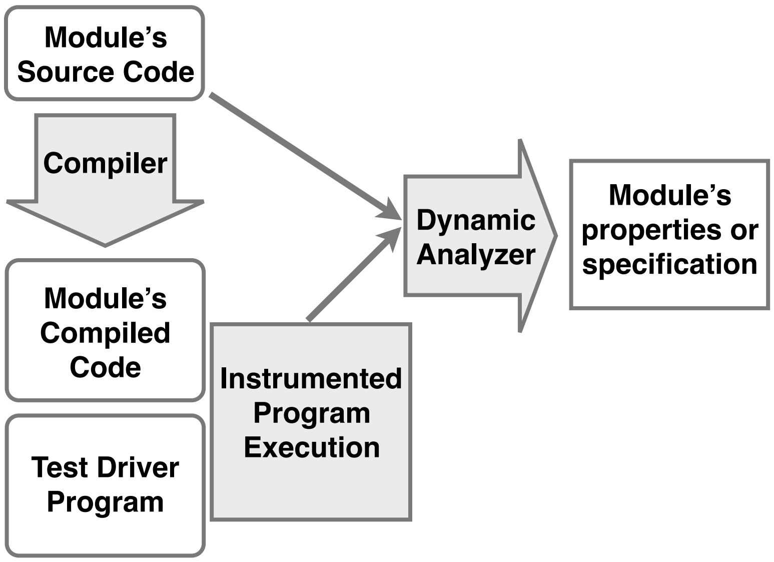


Figure 1.2. Dynamic Analysis considers the executing code of a targeted module, observing the transient values of variables and other metrics, and inferring properties from these. A dynamic analysis may also use static analysis to name, organize and interpret its output.

We assume that we can logically separate the role of instrumenting a module to emit trace streams from the role of analyzing the resulting trace streams. In fact, the Daikon analyzer separates these into separate executable processes, enabling repeated analysis of the same run. A schematic of dynamic analysis using separate collection and analysis processes is in Figure 1.3 below.

1.3. Dynamic Invariant Detection

Our research focuses on *dynamic invariant detection*, which observes a running program and infers specifications for targeted modules. In the literature, the term *invariant* is often used to refer to a clause (precondition or postcondition) in one of these inferred specifications. The Daikon dynamic invariant detector [Ernst et al., 2007] is a well-known representative of a particular class of invariant detectors (we include DIDUCE [Hangal and

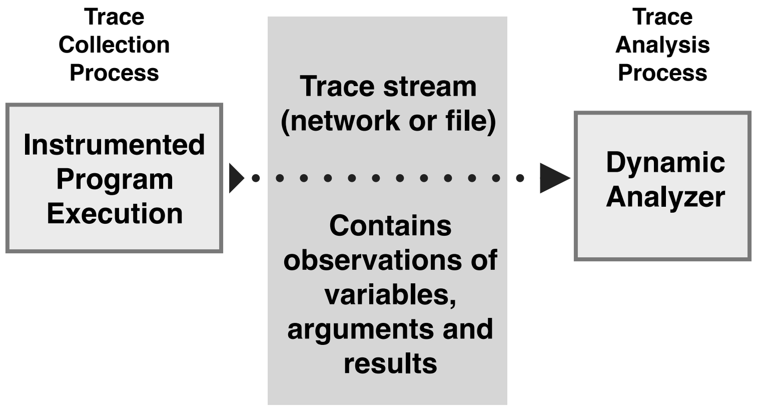

Trace Collection Process
Instrumented Program Execution
Trace stream (network or file)
Contains observations of variables, arguments and results
Trace Analysis Process
Dynamic Analyzer

Figure 1.3. Dynamic Analysis can be viewed as a collection process and an analysis process. These processes communicate via a trace stream containing runtime observations.

Lam, 2002] and Agitator [Boshernitsan et al., 2006] in this class) that obtain trace data from function entry and exit.

Daikon's approach to invariant detection can be viewed as a search problem, although there are several important optimizations in Daikon to make this search practical [Perkins and Ernst, 2004]. Daikon searches over a space of possible invariants, eliminating those that are falsified by evidence gathered during program execution. This is detailed in Chapter II. Whatever invariants remain at the end of a suitably robust execution are *presumed true* and can be considered as part of a potential specification. At the very least, the remaining invariants are consistent with the observed executions, which may reveal bias or weakness in the execution framework.

A simple example may clarify this. Consider the ToyMath module from 1.1. and an implementation and test program for it in Listing 1.1. For this example, we only include the sqr function.

There are an infinite number of potential candidate postcondition expressions that can be generated from the lexicon consisting of the argument, n, and the result value, return, as well as the arithmetic operations and common constants such as 0, 1, et al. Daikon populates its initial candidate set with a subset of these possible expressions; this subset is typically limited in complexity to expressions with one or two arithmetic operators and a single relational operator. In Table 1.2, we see a subset of those candidate invariants associated with the postcondition of sqr. Notice that not all of them can be true, and that several of them are likely to be falsified with only a single invocation of sqr. However, some of the candidates searched will be true for all of the invocations, and will be reported.

Table 1.2. A partial list of the candidate invariants for the ToyMath.sqr postcondition; these are all potentially true until falsified by execution observations.

return > n	return == n + 1
return * n == 0	return == 5
n == 0	n != 0
... and many more ...	

An analyst wishing to understand the sqr function will create a test program to exercise the function over a variety of values, and will execute this test program in conjunction with Daikon. Daikon will instrument the program so that traces are emitted at the entry and exit of the sqr function (as well as for all other public functions in the program, by default). At the end of execution, Daikon will report those invariants that are consistent with the invocations of sqr that were observed (Table 1.3). Prior to reporting invariants, Daikon filters out redundant and obvious invariants, which reduces the amount of invariants reported, usually to the benefit of the analyst.

As this example shows, a dynamic invariant detector can prove useful in characterizing a module's behavior in an easily understood form. In this example, the implementation was quite simple. The important point is that *any* implementation that satisfies our notion of a sqr function will result in the same invariants; the detector abstracts the implementation as an implementation-independent specification. This is potentially a very powerful tool, as we shall show in Chapter II, where we review some of the uses of a dynamic invariant detector.

Table 1.3. Daikon-reported invariants for the ToyMath.sqr postcondition.

return >= 0	return == n**2	return >= n

One troubling aspect that is hinted at in Table 1.3 is the need for us to ignore the true, but irrelevant, invariants in order to see the important ones. For example, the invariant return == n**2 is the only invariant of the three reported that usefully characterizes the implementation of sqr. The return >= 0 is implied by the laws of real-valued arithmetic, and is therefore not useful or interesting. On the other hand, the return >= n is a consequence of a weakness in our test: we are not exercising sqr with values between -1 and +1. We will discuss this latter aspect of using invariant detection to reveal test weaknesses in Chapter II.

As we increase the number of arguments, and add state variables to the lexicon of symbols available for invariant expressions, we dramatically increase the number of invariants that must be considered, and that are ultimately reported. The next section presents a more realistic example of applying invariant detection, and the associated difficulties.

1.4. Daikon and Alembic Comparison

One benefit of Daikon and similar tools is that they can automatically infer and report specifications. Unfortunately, they also generate a lot of distracting noise in the form of true, but irrelevant, statements; and they may also fail to detect important aspects of the behavior. In practice, a user must augment their use of dynamic invariant detection with filters and command line arguments in order to constrain the output to produce useful, focused, and relevant invariants. Our research explores the potential of using explicit abstractions to refine, shape and focus the invariant detection process in a way that is not possible or convenient via a first-generation detector like Daikon.

To illustrate how explicit abstraction helps focus invariant detection, we present a motivating example that contrasts Daikon with Alembic. We will attempt to analyze a portion of the MinMaxPriorityQueue (MMPQ) class from the Google Guava library [Google, 2011]. This will allow us to contrast the usage and analysis of Daikon alone with that of Alembic (which uses Daikon as its invariant detector). The MMPQ class implements a priority queue data structure that supports the retrieval of a minimum or a maximum, and also supports an optional bound on the queue size. Guava is a library of utility classes used by Google for their Java-based products.

Note that we will not be using the source code of the Guava library for this analysis; only the compiled .jar files will be used. Daikon and Alembic are able to *see* the internal member variables of the class, although the public Javadoc documentation does not indicate their existence. The goal of our analysis might be to understand the class better, or to verify that documented behavior is accurate, or perhaps to verify that our test program has no bias or missing coverage cases.

Ordinarily when using an invariant detector, we would examine the behavioral description of an entire module and its methods. For the sake of brevity in this example, we will focus on the single method, offer(<E> element), that takes an element of generic type E and adds it to the min-max priority queue. We exercise the queue and associated offer() with the main program in Listing 1.2.

The intent of a good test program is to exercise the offer method systematically in a variety of different situations. This helps ensure that the resulting invariants describe actual characteristics of the offer method, and not simply artifacts of a biased test program. Our particular test program below is written for clarity, and not thoroughness. In Chapter IV, we perform analysis of MMPQ and other Guava classes by using the unit tests written for Guava release testing.

1.4.1. Daikon Analysis of MMPQ

When we apply the Daikon tool to an execution of the above program, targeting the MinMaxPriorityQueue, we obtain the output in Listing 1.3. For this introductory example, our comparison will focus only on the postcondition of offer, as indicated by the suffix :::EXIT on the method name at the top of the listing[2]. In Chapters II and IV, we will see how the precondition associated with a method can also be useful. Appendix B presents listings of the Daikon invariants reported for all the public methods in MMPQ.

It is likely that all of the invariants reported by Daikon are true, but they are very uninformative, and the few invariants that characterize the behavior of offer() are buried amidst the rest. Sifting through the output in Listing 1.3, we can find a few invariants that seem to be genuine and useful descriptions; Table 1.4 contains these invariants and their interpretation.

Table 1.4. Useful invariants for MMPQ.offer reported by Daikon, and interpretations of these invariants.

Invariant	**Interpretation**
this.size-orig(this.size) - 1 == 0	The this.size variable increases by one each time offer returns.
this.size <= size(this.queue[])	The this.size variable is always bounded by the size of the internal array queue[].
this.queue[this.size-1] == this.queue[orig(this.size)]	The pre-state last element of queue becomes the post-state second-to-last element.
orig(element) in this.queue[]	The element argument is somewhere in queue at function exit.

[2]The name com.google.common.collect.MinMaxPriorityQueue has been shortened to MMPQ in most listings, and the invariants have been indented for readability. Invariants involving the getClass() function have been filtered in our Daikon examples because they are filtered by default in Alembic, and are not useful for most cases.

A typical Daikon user faced with this output would begin fiddling with the rich set of Daikon command line options in an attempt to filter the invariants down into something useful. Sometimes this is effective, but there is often as much work involved in finding the right set of option settings as in performing a manual analysis of the source code. Note that the above invariant this.size-orig(this.size)-1 == 0 could have been expressed more clearly as this.size == orig(this.size)+1. A Daikon user has very little control over which form of expression is emitted; we treat this as a minor inconvenience that can be addressed with post-processing. The major problem is not the individual form of the expressions, but the overwhelming amount of expressions to consider in the output.

Ordinarily, a skilled Daikon user would use the rich set of configuration options to *tune* the detection process for a specific problem. This configuration information may be passed on the command-line or via a configuration file, but it needs to be specified to allow the analyst to narrow the search and focus the reported invariants into a set that is useful and reasonable. For the output above, we instructed Daikon to ignore invariant expressions containing the symbol getClass(), which our experience has shown to produce mostly irrelevant invariants; Alembic disables getClass() by default. The full shell script that we used to run the previous Daikon example is in Listing 1.4.

1.4.2. Alembic Analysis of MMPQ

Alembic was designed to address the above 'needle in a haystack' problem of sifting through invariants, many of them over opaque internal structures. Instead of performing invariant detection upon observations of the concrete implementation class, Alembic uses synthesized observations on a user-defined abstraction class as the basis of invariant detection. Alembic is applied to a problem by creating an Alembic source file; in this example, we will call it MMPQ.alembic (Listing 1.5), and then using the alembic command to process the file.

In short, this file instructs Alembic to run a program, MMPQTest, and to lift invocations of MMPQ.offer into synthetic invocations on the trait SortedObjects. The trait declaration defines a synthetic class upon which to perform invariant detection; the view defines when and how to transform concrete invocations upon MinMaxPriorityQueue into abstract invocations upon SortedObjects. This technique allows the traditional invariant detector (Daikon) to perform inference on a more focused, possibly higher-semantic level, set of variables and functions. We will explain this in detail below.

One concern the reader might have is whether the investment in crafting such an Alembic file is worth the trouble. Minimally, this file specifies the test program to run and associated build configuration information; this information is similar to what is needed to run the program via Daikon (Listing 1.4). In the case of Daikon, we must not only specify the program to run, but we usually specify several command-line options to filter and focus the Daikon output. With Alembic, the compilation and Daikon invocation are performed by the alembic command behind the scenes. For simple abstractions, the size of an Alembic file is comparable to the size of a script needed to run Daikon.

Alembic was created to simplify the process of performing invariant detection, so that experiments can be performed and adapted more easily. Alembic manages the phases of invariant detection that would ordinarily be relegated to a shell script (Listing 1.4). This requires that the MMPQ.alembic file in our example have declarations of where and how to build the program, where and how to execute the program, and any custom analysis or postprocessing required. These roles are performed by the program, execution, and analysis declarations in the Alembic file.

A program declaration specifies build parameters for compiling and instrumenting a target main program and indicates which traits and views to use. An execution declaration specifies runtime arguments to the program and an execution directory, and the analysis declaration can be used to specify any special post-processing desired.

The abstraction declarations trait and view provide the analyst with the opportunity to shape the invariant detection process. Depending upon the amount of control they want, the size and complexity of the Alembic file changes. However, even simple abstractions that require only knowledge of the public methods of a concrete class can provide insight not easily obtainable with a traditional detector. When the analyst wishes to explore more sophisticated abstractions, Alembic's simple language is sufficient to express these without requiring any additional constructs; instead, the analyst can focus on expressing their abstractions.

A trait describes an abstraction consisting of method and state variable definitions; this abstraction is intended to represent some facet of the underlying concrete object's behavior and state. In this example, we surmise that a priority queue can be viewed as an ordered sequence, which we express in the SortedObjects trait. The use clause indicates that we wish to copy selected method signatures from the concrete class MinMaxPriorityQueue into our trait definition.

In order to generate observations to *feed* our traits, we specify corresponding views that indicate which concrete invocations should be instrumented, and how the concrete variables are used to compute values for the corresponding variables in the trait. These views are declared in the same Alembic file as the traits they reference. The lift clause of the view specifies the concrete class to measure, as well as the trait that will represent the concrete class. The lift_state clause provides a code fragment that implements a *representation function* that transforms concrete state into the abstract state within the view's trait.

For this example, we declare a view named ToSortedObjects that will lift invocations of offer from the concrete MMPQ to the abstract SortedObjects trait. The lift_state code invokes the toArray() method on the concrete object, and sorts it. The resulting sorted array is stored into the abs variable which refers to a synthesized instance of SortedObjects.

When we obtain the output from Alembic (Listing 1.6), we can see what our embedded traditional detector is able to discover when performing inference upon our SortedObjects trait. Simply considering the number of reported invariants, the human cognitive load is lower when examining the Alembic output. More importantly, the characteristic behavior of the underlying priority queue is revealed concisely in the Alembic invariants. Rather than having to understand the relationship between concrete variables and the effect of offer upon these, we can consider the effect on our abstraction, which encapsulates the concrete variables via the view, presenting us with a model-oriented view of the class.

1.4.3. How Does Alembic Do It?

It may be useful in understanding the above example to know how Alembic works (this information is presented in more detail in Chapter III). After creating the MMPQ.alembic file, and preparing a suitable test program, the analyst uses the shell command alembic MMPQ.alembic to invoke the Alembic system. This results in a model program being synthesized from the concrete program by adding generated classes and instrumentation. The resulting Alembic-enhanced program is then executed via Daikon's Chicory tool, which collects the trace information generated by the Alembic instrumentation and writes it to a file, where it is subsequently analyzed by the Daikon detector. We can view the alembic command as a supervisor (Figure 1.4) that first translates a program into a model program, then executes it to produce a trace stream reflecting model behavior, which is then analyzed by a traditional detector to produce invariants on the model.

The files that are generated are detailed in Table 1.5. The augmentation adds generated classes and AspectJ aspects which are responsible for:

- Representing the abstraction as a Java class.
- Instrumenting the targeted concrete executions.
- Invoking lift_state to translate concrete state to abstract state.

Table 1.5. Intermediate files generated from MMPQ.alembic, and their content and function.

Filename	Description
Abstraction_SortedObjects.java	This defines a class that contains any state variables declared for the abstraction, in this case, the variable sorted is our only state variable. This class also contains the necessary code to translate values and synthesize traces to the back-end detector (Daikon, in this case).
Adaptor_com_google_common_collect_MinMaxPriorityQueue.aj	This AspectJ aspect augments the compiled concrete class com.google.common.collect.MinMaxPriorityQueue so that executions of targeted public methods will be instrumented. When these methods are invoked, corresponding entry and exit methods are invoked on any views associated with this concrete class.
Abstractor_ToSortedObjects.aj	This aspect does not instrument any executions; rather, it augments the concrete class with the representation function, as well as any lift_method handlers. It is responsible for relaying observed concrete invocations from the Adaptor (above) to the Abstraction, possibly transforming values along the way.

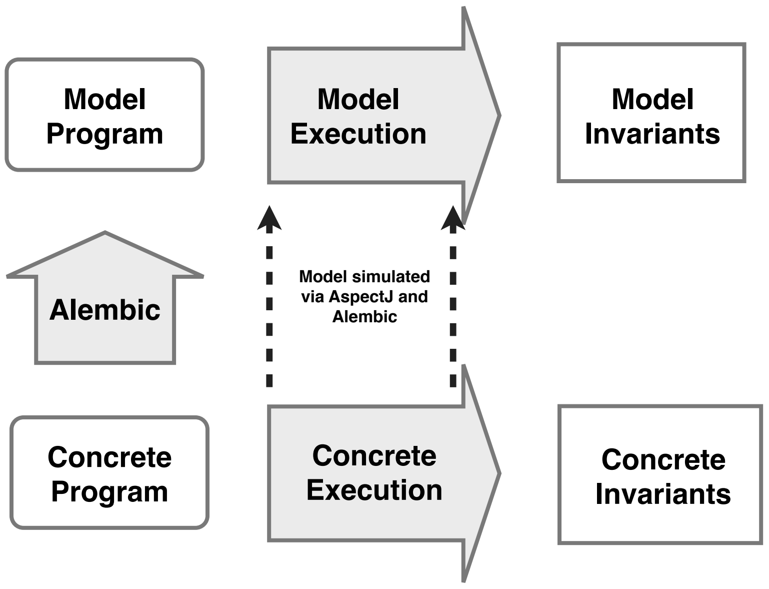


Figure 1.4. Alembic synthesizes an model program upon which invariant detection is performed.

1.5. Research Contributions

The central question of our research is whether dynamic invariant detection can be applied to user-defined abstractions of concrete modules, and the corollary questions of whether it is useful and practical. We answer affirmatively to all of these questions, and encourage the adoption of explicit abstraction features in existing and future dynamic invariant detectors. Our research develops ways to use existing detectors in conjunction with *explicit abstractions* to discover more useful and focused invariants. We introduce the Alembic system as a way to specify and apply these abstractions to existing programs and detectors, and to manage some of the mechanical tasks of post-processing and analysis. In addition to making dynamic invariant detection more useful, Alembic can expand the applicability of dynamic invariant detection to inferring program characteristics not previously considered as targets of such analysis. The main contributions of our work are summarized below.

1.5.1. Improving the Use of Existing Detectors

The use of explicit abstraction in conjunction with a traditional detector such as Daikon addresses some of the problems associated with such a tool. There are several aspects to this:

Managing complexity The limited depth and complexity of expressions considered by Daikon can be ameliorated by providing abstraction functions that encode complex expressions of multiple variables into expressions over one or more abstract variables, allowing the limited-depth inference mechanism to operate upon these derived abstraction expressions. This has the effect of constructing a more refined search space within which to search for invariants; in addition, invariants that were too complex or deep to be discoverable may now be considered because some of the complexity has been supplied by the explicit abstraction.

Reducing opacity The general-purpose nature of a detector like Daikon prevents it from discovering useful invariants when the underlying data structure is opaque or not expressible in one of the detector's canonical forms (String, List, Object, Integer, et al) in a meaningful way. This can be addressed by providing abstraction functions that lift these opaque or convoluted concrete implementation structures into more general structures and values that are amenable to traditional analysis. For example, a binary heap is an efficient structure for maintaining a sorted list, but its internal structure is inscrutable to an invariant detector, which will see it simply as an array of elements. Alembic provides a lift_state clause to translate this heap into a more transparent, but less efficient, sorted array, which may produce more meaningful invariants.

Restricting visibility Applying an invariant detector to a concrete implementation can produce a large amount of invariants, many of them true but irrelevant; others are based upon internal state or arguments that may not be of concern to an analyst (e.g., internal statistics variables) for a given problem. But if we apply our detector to an abstracted version of the implementation, we can control which variables and arguments are visible; this enables the focusing of the tool onto the areas in which the analyst is most interested.

Together, these features of abstraction-based invariant detection allow the lifting of a concrete implementation to a higher semantic level, where the resulting invariants may be more meaningful and understandable by a human analyst.

1.5.2. New Applications of Invariant Detection

Our research into abstraction was originally motivated by a desire to create more useful and meaningful invariants via Daikon; we soon discovered that the techniques enabled us to apply a traditional detector to a variety of entirely new problem domains. Below is a summary of some of the applications we have considered and partially explored:

State Abstraction The most basic form of abstraction is where the state of the concrete object is mapped to an abstracted state within the abstraction object. Variables in the concrete object may be converted to corresponding variables in the abstraction or may be simply ignored. Variables in the abstraction may be synthesized from multiple variables in the concrete object. The lift_state clause of an Alembic view specifies a representation function that lifts the concrete state to an abstracted state defined by a trait.

Effect Abstraction We can construct abstractions whose state is derived from *effects* that would ordinarily be unobservable by an invariant detector. For example, a File object's read and write methods could be abstracted such that the effect on the file system is captured as observable state on an Alembic trait. This in turn enables an invariant detector to perform inference on the read and write methods' effects. The lift_method clause of a view enables the capture and transformation of arguments and runtime properties (e.g., CPU time) by providing entry and exit hooks for the targeted methods.

Abstract Functions We can consider a sequence of statements in a Java program as a composition of functions (in the functional programming sense), each of which performs some computation and successively modifies the shared state. If we treat the start and end of such a sequence as the entry and exit of an abstract method in an Alembic trait, then our invariant detector can report invariants over the set-theoretic function that the composition implements. This feature currently requires the explicit insertion of Alembic *probes*, which are calls to liftEntry and liftExit methods on the generated abstraction class. This abstract function capability can be used to determine invariants over common sequences of method calls (e.g., f.open;f.read;f.close or s.push;s.pop) or, as we detail below, common control structures such as loops.

Loop Invariant Detection This is actually a special case of abstract functions, where we view the loop entry as the beginning of our composition, and the loop exit as the end. Applying an invariant detector to the resulting abstraction enables the discovery of the loop variant and invariants; similarly, an abstract function from before the loop to after the loop can be used to summarize the loop's behavior as a *black box*.

Aggregate Behavior Detection The methods of a module are often logically partitioned into different roles: e.g., readonly methods, private methods, insertion methods, and deletion methods. Given a module where can we can partition its methods into different logical groups, we can define an abstraction of the module that contains a parameterless function for each of these groups. By abstracting each concrete method into the shared method corresponding to its group, we enable an invariant detector to infer the pre and post conditions common to each member of a group. This may result in a better description of the module's behavior; for example, all of the read* methods in a File class could be grouped and a common precondition of isOpen==true might be deduced.

History Constraint Detection A specific usage of aggregate abstraction is where we aggregate the public methods into a single abstract method. The resulting invariants reported on this method correspond to the JML history constraint for the module [Leavens, 2006]. A JML history constraint is a set of pre- and post-conditions that must hold for all methods (or public methods) in the class or any of its subclasses. For example, a PatientVisitLog class might wish to ensure that records can only be added and that no subclasses can ever violate this; a history constraint is the appropriate way to specify this requirement.

1.5.3. Alembic System

We developed the Alembic program analysis system to allow the easy expression and application of abstractions. The system consists of a language, a runtime, and a workflow manager. These components are described below in terms of the features they provide, as well as a brief description of how they manifest in the current implementation.

Alembic Language The language allows for the specification of program, execution, trait and view constructs in an Alembic source file. program and execution entities provide a way to specify build and execution environments and configurations. trait and view entities are the means by which Alembic creates synthetic Java classes and functions that will be analyzed by the invariant detector. An Alembic source file is effectively defining an experiment to run, where the invariant detection is performed not on the concrete functions of the implementation, but on the synthetic classes and functions generated by Alembic. The Alembic compiler is an ANTLR-based parser [Volkmann, 2008] that generates a makefile that manages the invocation of other tools and code generators written in Python.

Alembic Runtime The runtime supports the synthesis of abstraction classes and the dynamic mapping of concrete method invocations into invocations on the abstraction. This results in execution traces on the concrete program being reflected and transformed into execution traces on the synthesized abstraction classes, where they can be analyzed by an ordinary dynamic invariant detector (Daikon, in the current implementation). Alembic achieves this by using AspectJ to modify the target module by inserting calls to abstraction and trace-generation code at the entry and exit of each targeted concrete method.

Build Management We designed Alembic with a scientific workflow viewpoint, where we consider the concrete execution of a target program and its subsequent analysis to be a series of tasks that are mechanical and can be mostly hidden from the analyst. The refined analysis results at the end of this pipeline are what we seek for most purposes. Another benefit of this model is that we can add additional analysis tools or invariant detectors within the pipeline without requiring a change to the Alembic files describing the experiments. We have found that the isolation of the various command line arguments and build phases from the specification of the desired abstractions has made it much easier to perform experiments than the traditional way of maintaining a shell script to invoke the various phases directly.

Alembic makes some things easy that would otherwise be difficult, error-prone or tedious:

- Different experimental abstractions can be tried out without modifying the source program.
- The Alembic language provides a convenient high-level way to express abstractions and the relation between these abstractions and the underlying concrete modules.
- Reusing traits and views is easier because they are specified separately from the concrete execution.
- Tuning, parameters, and post-processing that would normally be necessary when using a raw invariant detector are encapsulated in the Alembic build system, enabling the analyst to focus on experimenting with different abstractions and obtaining useful results.

1.5.4. AspectJ-based Instrumentation

Aspect-oriented programming (AOP) is a programming language facility that allows new code to be *woven* into existing sources or compiled code in a systematic, structured way. The new code and its weaving directives are specified as one or more *aspects*, when are then applied to existing code by an aspect compiler or code weaver. AspectJ [Kiczales et al., 2001] provides AOP for Java; AspectC++ [Spinczyk et al., 2005] provides a similar facility for C++. Although our current work focuses primarily on Java and AspectJ, the results are applicable to any other language with an AOP facility.

One of the important features of Alembic is that it enables traits and views to be declared in an Alembic source file; these are compiled into Java classes and AspectJ aspects. The aspects are *woven* into one or more concrete classes as specified in the Alembic file. These aspects add instrumentation to the concrete classes so that every concrete method execution is wrapped with code that emits a pair of traces corresponding to the pre-invocation state and post-invocation state. These traces are not the traces of genuine concrete method invocations, but are instead synthetic traces on the corresponding abstraction.

Importantly, the Alembic file may specify executable code in its view definitions that is intended to transform or filter the data from the concrete invocations. This code executes in the context of the running program, and not in the context of the analyzer, which is often a separate process that reads trace files at a later time.

Ordinarily, a Daikon user would use the Chicory tool (Daikon's Java bytecode instrumenter) to instrument their code with calls to the Chicory trace-generation functions. This instrumentation is achieved by a BCEL (Byte Code Engineering Library) API that modifies the Java classes at the byte code level [Dahm, 2001]. Chicory instruments every public method entry and exit, by default; this can be adjusted via command-line options.

Our original experiments in applying abstraction techniques to dynamic invariant detection involved a modified version of Chicory. Modifying this was difficult and the result was inflexible. We abandoned this effort and instead chose to use AspectJ to weave our instrumentation into target classes. This has enabled rapid progress and suggested further enhancements such as context-sensitive invariant detection.

In the current version of Alembic, AspectJ serves as the instrumenter, and the woven aspects first invoke the Alembic abstraction mechanisms which will ultimately result in the invocation of the Chicory trace-generation functions to generate the synthetic traces. The use of AspectJ has several advantages over the use of Chicory:

Flexibility AspectJ allows us to specify arbitrary code to execute at the concrete execution entry and exit points. We use this to transform the concrete visible variables into a trace tuple that will be emitted to the invariant detector for analysis. We suspect that the potential for this flexibility is mostly untapped in our current implementation, and look forward to using this to implement more sophisticated abstractions and inference mechanisms.

Independence By using AspectJ instead of Chicory, we obtain a solution that is more independent of the particular invariant detector. In the event that we want to adapt Alembic to work with a non-Daikon detector, it is far easier to adapt Alembic's aspect generation code to the new detector, rather than trying to modify Chicory even further. Most likely, we would end up abandoning Chicory and rewriting an instrumenter from scratch; fortunately, the use of AspectJ avoids the entire problem.

Control Daikon has a very limited way of controlling what gets analyzed and when. Command-line arguments provided at startup govern the entire program execution. Modifying these parameters during execution is possible, but requires modifying the target program's source code. AspectJ lets us control when, where and in what context our instrumentation runs, and it does so with a very fine degree of control. Importantly, we can augment the concrete program with abstractions without requiring that it be rebuilt from source. We anticipate that this fine control will be useful as we develop more sophisticated abstractions, or when we want to be very selective about what gets analyzed and when.

1.6. Structure of this Document

This Chapter I, Introduction, presented dynamic invariant detection as a technique for use in software analysis and specification, and some of the problems with the technique. We showed how applying invariant detection to a model program abstracted from the concrete can address some of these problems by allowing an analyst to specify explicit abstractions that are then used as targets of dynamic invariant detection.

In Chapter II, Static and Dynamic Techniques, we begin with a review of the background and terminology used in the software analysis and specification field. We will focus on the Daikon system for dynamic invariant detection, which is the mechanism we build upon for Alembic. We look at dynamic invariant detection and several alternative techniques that have been used in its implementation.

Chapter III, Abstraction and Alembic, details the principle of explicit abstraction applied to dynamic invariant detection, and presents the Alembic language as a way to facilitate the use of this technique. We will describe the syntax, usage and architecture of Alembic.

Chapter IV, Examples, Experiments, and Results, develops a set of examples that illustrate the capabilities of Alembic. These include proofs-of-concept to illustrate particular abstraction patterns, as well as richer examples applied to production libraries such as the Guava Collection classes.

Chapter V, Conclusion, summarizes the current state of the project and suggests promising directions and work. We also briefly describe some of the avenues exposed, but not fully explored, by this research. We offer the results of our research to inform next-generation dynamic invariant detectors; we believe that the features offered by abstraction can be fruitfully integrated into the software analysis tool suite.

Some of the research described in this dissertation was performed in collaboration with Michal Young, who participated in the refinement of the abstraction theory and its explication. In addition, Michal Young and Yannis Smaragdakis collaborated on the preparation and submission of a conference paper for FSE 2012. I was the primary contributor to the theory of abstraction, developed the Alembic language, and did all of the writing for this dissertation.

Listing 1.1. The ToyMath Java class, and a main to exercise it.

```
public class ToyMath
{
  public float sqr( float n )
  {
    return n * n;
  }

  public static void main( String[] args )
  {
    ToyMath tm = new ToyMath();

    for ( int i = 0; i < 10; ++i )
    {
      for ( int j = 0; j < 100; ++j )
      {
        tm.sqr( j );
      }
    }
  }
}
```

Listing 1.2. This test program, MMPQTest, exercises the MMPQ.offer method with positive, zero and negative integers.

```
import com.google.common.collect.*;

public class MMPQTest
{
    public static void main(String[] args)
    {
        MinMaxPriorityQueue.Builder<Comparable> builder =
            MinMaxPriorityQueue.maximumSize( 1000 );

        for ( int whichIter = 0; whichIter < 20; ++whichIter )
        {
            MinMaxPriorityQueue<Integer> b = builder.create();

            for ( int i = 0; i < 10; ++i )
            {
                b.offer( i );
            }
            for ( int i = 0; i < 10; ++i )
            {
                b.offer( -i );
            }

            for ( int i = 0; i < 15; ++i )
            {
                b.removeFirst();
            }

            for ( int i = 1; i < 10; ++i )
            {
                b.offer( i * i );
            }
        }
    }
}
```

Listing 1.3. Invariants reported by Daikon for the postcondition of MMPQ.offer() when exercised by the MMPQTest driver program. Compare to Listing 1.6.

```
MMPQ.offer(java.lang.Object):::EXIT
  this.minHeap == orig(this.minHeap)
  this.maxHeap == orig(this.maxHeap)
  this.maximumSize == orig(this.maximumSize)
  this.queue[this.size−1] == this.queue[orig(this.size)]
  this.size >= 1
  this.modCount >= 1
  return == true
  this.queue[this.size−1] != null
  this.maximumSize > orig(this.size)
  this.maximumSize > orig(this.modCount)
  this.maximumSize > orig(size(this.queue[]))
  orig(element) in this.queue[]
  this.size − orig(this.size) − 1 == 0
  this.size != orig(this.modCount)
  this.modCount > orig(this.size)
  this.modCount − orig(this.modCount) − 1 == 0
  MMPQ.EVEN_POWERS_OF_TWO > orig(this.size)
  MMPQ.EVEN_POWERS_OF_TWO > orig(this.modCount)
  MMPQ.ODD_POWERS_OF_TWO < orig(this.size)
  MMPQ.ODD_POWERS_OF_TWO < orig(this.modCount)
  this.queue[MMPQ.DEFAULT_CAPACITY−1] in orig(this.queue[])
  orig(this.size) <= size(this.queue[])−1
  orig(this.modCount) != size(this.queue[])
  size(this.queue[]) >= orig(size(this.queue[]))
  size(this.queue[])−1 != orig(size(this.queue[]))
  size(this.queue[])−1 >= orig(size(this.queue[]))−1
```

Listing 1.4. Shell commands required to compile MMPQTest, instrument and execute it with Guava's MMPQ, and to analyze the results with Daikon.

```
# Compile the unit test. MMPQ is already compiled in the Guava lib
> javac −g −cp $CLASSPATH:${GUAVA_JAR} MMPQTest.java

# Execute an instrumented (via Chicory) MMPQTest, collecting traces
> java \
      −cp .:$CLASSPATH:${GUAVA_JAR} \
      daikon.Chicory \
      −−ppt−select−pattern='com.google.common.collect.[^.]+.offer' \
      −−omit−var='getClass' \
      −−dtrace−file=MMPQTest.dtrace.gz \
      MMPQTest

# Interpret the traces and report invariants detected
> java \
      daikon.Daikon \
          −−var−omit−pattern='getClass' \
          −−config_option daikon.Daikon.progress_delay=−1 \
      MMPQTest.dtrace.gz
```

Listing 1.5. The MMPQ.alembic file contains declarations required to lift MMPQ.offer invocations into corresponding invocations on the SortedObjects trait, via the ToSortedObjects view.

```
trait SortedObjects
[
    use com.google.common.collect.MinMaxPriorityQueue : "^offer$";
    state Object[] sorted;
]

view ToSortedObjects
[
    lift com.google.common.collect.MinMaxPriorityQueue<?> : "^offer$" to
        SortedObjects;
    lift_state
    {
        java.util.Arrays.sort( abs.sorted = this.toArray() );
    }
]

program MMPQP
[
    builddir "../../examples/MMPQ/";
    views "ToSortedObjects";
]

execution MMPQPE
[
    program MMPQP;
    rundir "../../examples/MMPQ/";
    main "MMPQTest";
]

analysis MMPQPEX1
[
    execution MMPQPE;
]
```

Listing 1.6. Alembic-reported postconditions for SortedObjects.offer as abstracted via the ToSortedObjects view of the concrete MMPQ.offer method.

```
SortedObjects.offer(java.lang.Object):::EXIT
  size(this.sorted[])−1 == orig(size(this.sorted[]))
  return == true
  size(this.sorted[]) >= 1
  orig(element) in this.sorted[]
```

CHAPTER II

STATIC AND DYNAMIC ANALYSIS

This chapter presents some background on the software analysis concepts and techniques that are related to the motivation, implementation and understanding of our research. We begin by reviewing the relevant terminology of software specification and analysis, and look at techniques whose goal is to infer, communicate and enforce specifications.

Our discussion of specification will begin with Hoare's Axiomatic Semantics [Hoare, 1969], its adoption as a behavioral specification mechanism with the contract-based Eiffel programming language [Meyer, 1992] and the two-tiered Larch Specification Language [Guttag et al., 1985], and its eventual embodiment in the Java Modeling Language (JML) [Leavens, 2006]. We will discuss the syntax of JML and some of its applications, especially those related to invariant detection. We will describe the Larch family of specification languages and the influence of Larch on JML's *model field* capability and on the idea of explicit abstraction and Alembic.

We then describe and contrast static and dynamic analysis techniques that are related to the specification ideas above. Static techniques include symbolic execution, abstract interpretation, flow analysis, and other forms of reasoning that can be performed on the source code without requiring execution. For example, most compiler optimizations are the result of various static analyses. We briefly discuss the *halting problem* and its relevance to static and dynamic analysis.

Finally, we present dynamic analysis techniques, their applications, and the additional benefits they can provide over static analysis. A dynamic analysis is a method for *measuring* dynamic properties of executing programs, and *inferring* understandable and useful information from these measurements.

We survey several dynamic techniques, including dynamic symbolic execution, profiling, and tracing. We then present dynamic invariant detection in detail, explaining Daikon's approach and contrasting it with other ways to approach the problem of invariant detection.

2.1. Behavioral Interface Specification Languages

The field of software specification encompasses languages and systems that characterize the structure, behavior and constraints on software entities. These systems describe large-scale system concepts such as process and system architecture, data flows, performance and resource metrics, security, and protocols. They also describe smaller scale entities such as software modules, classes and functions; it is this area of specification where we will be concentrating our discussion in the following sections.

To explain our research and place it into context, we will primarily be presenting the particular set of specification languages and applications that are related to module and function specification, dynamic invariant detection and abstraction. The particular class of specification languages we will focus on are called *Behavioral Interface Specification Languages*, or *BISL*s [Leavens et al., 1998].

A BISL specifies two aspects of a module:

Interface This is the structure of a module in terms of its publicly visible methods, features and supported interfaces. Often, the interface specification simply refers to the implementation language's syntax for this structure and provides additional details that are not part of the implementation syntax.

Behavior This is a formal description of how the module behaves when used, usually by providing clauses for each method as well as the module itself. These clauses clarify the external semantics of each method, as well as any module invariants that must hold outside of methods. Most modern implementation languages do not support behavior specification natively, and the specification is provided as an annotation or in a separate file. Eiffel, of course, embeds the behavior specification syntax natively in the language (See 2.1.2. Eiffel and Design By Contract).

A BISL is distinct from an Interface Definition Language (IDL), which usually specifies the methods and argument types of an *interface*, but has no provisions for specifying *behavior*. It is also distinct from an ordinary programming language such as Java, where a class implementation defines a particular interface and behavior, but where this behavior is not summarized into an externally usable specification. Eiffel is unusual in that it is a programming language that also contains a way to specify behavior in clause form as part of the externally usable interface.

There are a multitude of specification languages and means to enforce, verify, and infer them. For this dissertation, we will focus on a particular line of descent that culminates in Java Modeling Language, invariant detection and the ideas of abstraction presented here. In Figure 2.1, we see how the earlier technology and theory influenced the present-day Java Modeling Language.

The invariants reported by a dynamic invariant detector are a form of specification. As we described in the Introduction, a specification can be viewed as some form of description that constrains implementations. Different formalizations of specifications emphasize

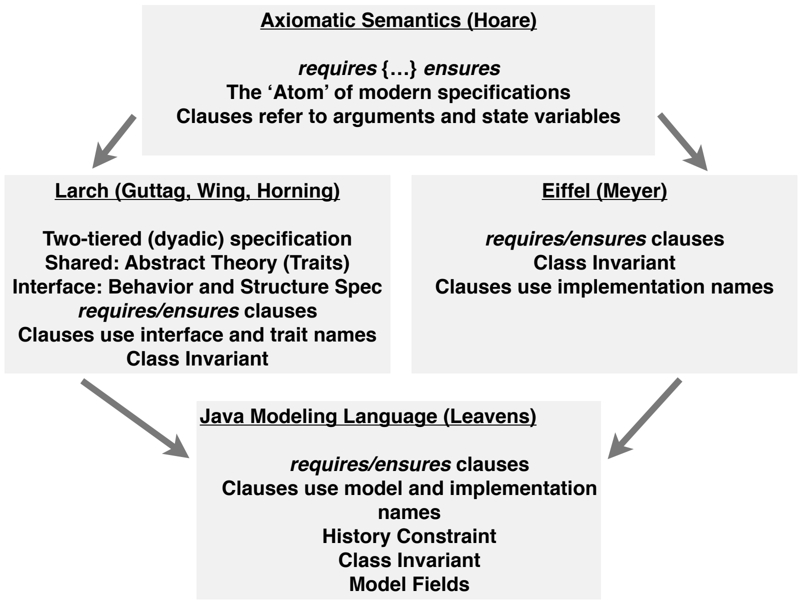

Axiomatic Semantics (Hoare)
requires {...} ensures
The 'Atom' of modern specifications
Clauses refer to arguments and state variables
Larch (Guttag, Wing, Horning)
Two-tiered (dyadic) specification
Shared: Abstract Theory (Traits)
Interface: Behavior and Structure Spec
requires/ensures clauses
Clauses use interface and trait names
Class Invariant
Eiffel (Meyer)
requires/ensures clauses
Class Invariant
Clauses use implementation names
Java Modeling Language (Leavens)
requires/ensures clauses
Clauses use model and implementation names
History Constraint
Class Invariant
Model Fields

Figure 2.1. Behavioral Interface Specification Languages (BISLs).

different properties that are to be constrained and described. In the following sections, we look at some of these formalizations and how they relate to dynamic invariant detection.

2.1.1. Hoare Triples

One of the earliest attempts to characterize program semantics was Hoare's Axiomatic Semantics [Hoare, 1969], which specified a way to assign meaning to programs, procedures and statements. The semantics consists of formulae called *Hoare triples*, and a set of axioms and inference rules for generating these triples. A Hoare triple is of the form: P {S} Q, where P is the *precondition*, Q is the *postcondition*, and {S} is a statement, procedure or program. The interpretation of this triple is:

> When {S} begins execution in a state where P holds, then Q will hold at the end of the execution of {S}. If {S} does not terminate, then Q does not necessarily hold (partial correctness).

Using the axioms and inference rules, it is possible to prove triples for sequences of statements, procedures and whole programs. Our interest in Hoare triples stems not from the inference rules and proof theory, but from the idea that the precondition and postcondition characterize the possible behavior of the associated program {S}, while not mandating a particular implementation. This idea forms the basis of many subsequent specification languages, including Eiffel, Larch, and JML, which we describe in the following sections; it is also the basis of how dynamic invariant detectors structure and report discovered specifications.

An example of a Hoare triple describing the function sqr is in Listing 2.1, where we see the sqr function defined.

Listing 2.1. A Hoare triple characterizing the ToyMath.sqrt function, written vertically for readability, and to suggest the form used by textual specification languages.

```
n >= 0
{ result = ToyMath.sqrt( n ) }
return**2 == n
```

The precondition and postcondition of a Hoare triple are logical clauses over arguments and state variables, and a natural interpretation of the triple is as a *software contract*. If the caller ensures that the precondition is satisfied, then the body of the triple will guarantee that the postcondition is satisfied upon exit. If we elide the body of the triple and only consider the precondition/postcondition pair, then we have isolated a *specification* of a function that is independent of its implementation. This is the basis of all of the specification languages we will discuss in this chapter, as well as forming the basis of dynamic invariant detection. This idea of software contract will be developed further in the following sections with discussions of Eiffel, Larch and JML.

An alternative interpretation of Hoare semantics considers the code of a triple to be the implementation of corresponding set-theoretic function from one set of *possible worlds* (the domain) into a different set of possible worlds (the codomain). The domain is defined as the set of variable value tuples that satisfy the precondition of the triple; similarly, the codomain is defined as the set of variable value tuples that satisfy the postcondition. In Figure 2.2, we illustrate the correspondence between a Hoare triple specification and a functional specification. We use the notation sat(pre) and sat(post) to indicate the sets of states that satisfy the logical clauses in pre and post, respectively.

The advantage of this interpretation for our purposes is that it is a better model for dynamic invariant detection, which is based upon capturing these value tuples and then characterizing them as a set of logical clauses. In a sense, the dynamic invariant detection problem is about *reverse engineering* the Hoare triple precondition and postcondition clauses by observing value tuples emitted before and after the triple's implementation executes. From the sets sat(pre) and sat(post), dynamic invariant detectors infer pre and post clauses. This will be expanded upon later in this chapter (see Dynamic Invariant Detection below).

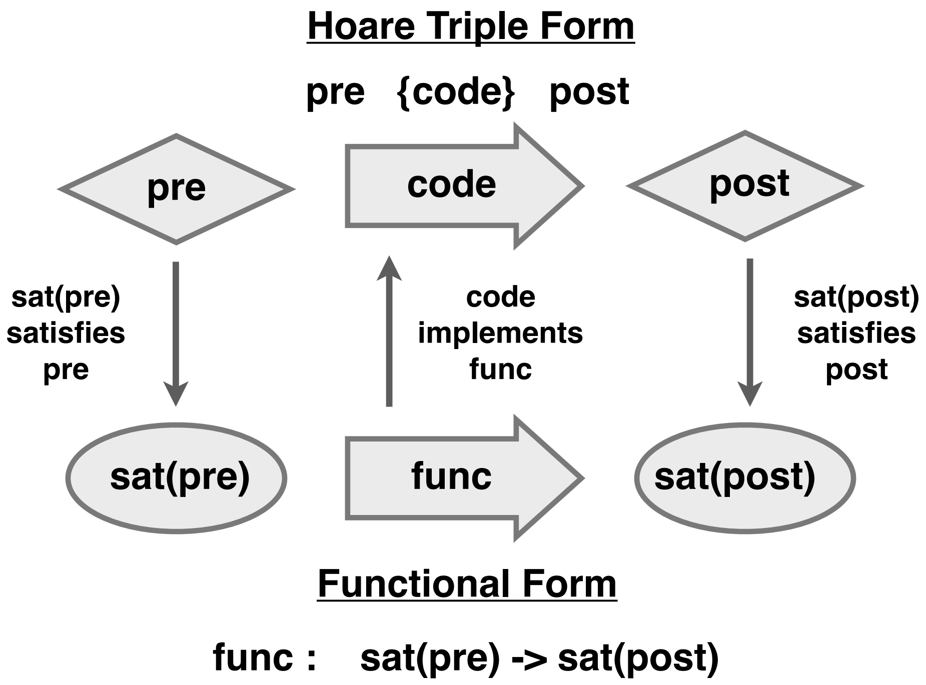


Figure 2.2. A Hoare triple as a set-theoretic function function from sat(pre) to sat(post). sat(pre) and sat(post) indicate the sets of states that satisfy the logical clauses pre and post, respectively.

The functional interpretation enables one to more easily imagine exotic applications of dynamic invariant detection where we are considering the code of the triple to be something other than an ordinary procedure body. In the case of the abstraction techniques we discuss in Chapter III, this code does not even exist, it is an illusion created by synthesizing traces to the invariant detector. We extend this even further in Chapter IV, where we synthesize functions from loop bodies or from aggregate behavior of the methods in a module. As long as we can populate the sets sat(pre) and sat(post) with tuples corresponding to some abstract function, then the invariant detector can infer pre and post invariants from these tuples.

2.1.2. Eiffel and Design By Contract

The Eiffel programming language ensconced the idea of Hoare triples in the syntax of its module definitions via the use of require (precondition) and ensure (postcondition) clauses associated with each function in an Eiffel class. Eiffel was designed to enable the development of large systems with boundaries defined by behavioral interface specifications, and to facilitate a software methodology known as *Design-by-Contract* [Meyer, 1992]. Eiffel is unusual in that the use of behavioral specifications is built into a programming language syntax, rather than as a specification-only language or annotation language (see Larch Specification System and Java Modeling Language (JML)

We can implement our example ToyMath module from earlier in this chapter (Table 1.1) as an Eiffel class; a client of this class would likely consult the corresponding class interface (Listing 2.2) as the behavioral specification for the class. Eiffel uses the keyword feature to indicate both functions and attributes within a class. The require clause indicates that a necessary precondition to calling sqrt is that the input argument, n, is positive. The ensure clause indicates that the result of the function, indicated with Result, can be squared to produce the input, n.

Listing 2.2. An Eiffel interface specification for a TOYMATH class.

```
class interface
    TOYMATH

feature
    sqrt ( n : DOUBLE ) : DOUBLE
        require
            n >= 0
        ensure
            Result * Result = n

    sqr( n : DOUBLE ) : DOUBLE
        ensure
            Result = n * n

    floor( n : DOUBLE ) : INTEGER
        ensure
            Result <= n
            n − Result < 1

end −− class ToyMath
```

Our interest in Eiffel is in the use of require and ensure to characterize the expected behavior of functions, and the use of these *software contracts* as specifications. Eiffel also introduces the idea of a *class invariant*, which is a set of clauses which must hold when a method of the class is not in execution; in other words, the class invariant holds before execution of a method and after execution of a method. In effect, the class invariant is conjoined with the the precondition and postconditions for every method.

Although Eiffel is a programming language, it has all of the important aspects of a behavioral interface specification language (BISL), including the declaration of preconditions, postconditions and object invariant clauses. These clauses are expressions phrased in the concrete Eiffel language and referring to members and functions belonging to concrete Eiffel classes. In the next section, we contrast this concrete form of behavioral specification with the two-tiered system of Larch, which provides for abstracted specifications.

2.1.3. Larch Specification System

We have described how Hoare specifications characterize the behavior of a function by describing the precondition and postcondition as logical clauses over the visible implementation variables and arguments. We have shown how Eiffel incorporates this notion of software contract into a full programming language that characterizes the structure and behavior of an interface as a set of per-method contract-style specifications, as well as a class invariant. We now look at Larch [Guttag et al., 1985], which is not a programming language, but is instead a behavioral interface specification language intended to be used in conjunction with a traditional programming language such as C++ [Leavens, 1996]. Unlike non-behavioral IDLs (Interface Definition Languages), Larch provides a way to specify the behavior of an interface as well as its structure.

Larch was influential in the development of subsequent specification languages and features, including the Java Modeling Language, which we will discuss in the next section. Larch is interesting for our purposes because abstraction plays a central role in Larch; specifications relate language-specific interface details such as methods, arguments and results to abstractions known as traits, which have no implementation and are specified in an algebraic specification language (the Larch Shared Language).

Larch is known as a *dyadic* or *two-tiered* specification language because a specification is separated into two formal languages, one focused on expressing an abstract *theory* of a data type, and another focused on binding that theory to a particular implementation [Guttag et al., 1993]. The Larch Shared Language (LSL) provides a way to specify abstract traits consisting of operators and their signatures, and equations that relate these operators. Alembic's notion of abstraction was inspired by Larch, and we have adopted Larch's trait keyword as the way that abstractions are declared in an Alembic source file.

The Larch system is designed to express a reusable library of commonly used concepts and data structures in an implementation-independent algebraic language, the Larch Shared Language (LSL). Examples of traits in this specification library include analogues of common data structures such List, PriorityQueue, and MultiSet, as well as mixin-style traits such as Enumerable and TotalOrder. A Larch trait contains no implementation; instead, it consists of operators and a set of algebraic expressions that relate these. The value of the algebraic shared language is that it allows reuse of the algebraic abstractions across multiple implementation languages, and it also is amenable to theorem proving. We provide an example of an LSL definition of a priority queue trait in Listing 2.3.

To complement the language-independent *algebraic* Shared Language, Larch provides an *operational* interface language for each particular implementation language to be used (for example, Larch/SmallTalk and Larch/C++). The role of specifications written in the interface language is to describe the implementation's interface in terms of one or more traits in the LSL. For example, the interface specification for a C++ PriorityQueue class would be written in Larch/C++ to provide a behavioral specification of the class's methods in terms of the abstract trait PriorityQueue, which is itself specified in the LSL.

Larch/C++ is implemented as an annotation language upon ordinary C++. This way there is only one syntax for the *interface* aspects and a separate syntax for the *behavioral* aspects. We provide a fragment of a Larch/C++ specification of a priority queue in Listing 2.4. Note that LSL traits are stateless, but that LIL specifications usually assume stateful underlying implementations.

A C++ class implementation that satisfies a LIL specification will provide methods that correspond to those in the specification, and these methods will behave according to the specification. This behavior is described in a contract-style, with logical requires and ensures clauses for each method. The LIL specification acts as a *bridge between* the language-specific concrete implementation and the algebraic theory of the underlying datatype. Alternatively, the LIL specification acts as a *view* of the concrete implementation as seen through the lens of one or more LSL traits.

In the sections above, we considered the Hoare specification of a procedure as a function from sat(pre) to sat(post). We showed how we can consider an Eiffel specification as a structure containing one of these functions for every method. In both of these cases, the names used in the behavioral clauses are drawn from the names visible from each function (arguments, results and state variables). Larch is different from these systems because the behavioral clauses are expressed in terms of names drawn from the traits written in the shared language.

Listing 2.3. A Larch Shared Language specification for a PriorityQueue trait (from Leavens [1996]).

```
PriorityQueue ( >:E, E −> Bool, E, C): trait
assumes TotalOrder (E for T)
includes Integer
introduces
    empty: −> C
    add: E, C −> C
    count: E, C −> Int
    __ ∈ __: E, C −> Bool
    head: C −> E
    tail: C −> C
    len: C −> Int
    isEmpty: C −> Bool

asserts
    C generated by empty, add
    C partitioned by head, tail, isEmpty
    ∀ e, e1: E, q: C
        count(e, empty) == 0;
        count(e, add(e1, q)) == count(e, q) + (if e = e1 then 1 else 0);
        e ∈ q == count(e, q) > 0;
        head(add(e, q)) ==
            if q = empty ∨ e > head(q) then e
            else head(q);
        tail(add(e, q)) ==
            if q = empty ∨ e > head(q) then q
            else head(q);
    len(empty) == 0;
    len(add(e, q)) == len(q) + 1;
    isEmpty(q) == q = empty

implies
    Container (add for insert)
    ∀ e, e1, e2: E, q: C
    add(e1, add(e2, q)) = add(e2, add(e1, q));
    len(q) ≥ 0;
    add(e, q) ≠ empty
    converts count, ∈, head, tail, len, isEmpty exempting head(empty), tail(empty)
```

Listing 2.4. Larch/C++ PriorityQueue specification. Suffixes ˆ and ’ refer to the pre- and post- values of the suffixed variables, respectively (from Leavens [1999]).

```
template <class Elem /*@ expects PriorityQueueRequirement(Elem) @*/>
              //@ where Elem is {
              //@ bool operator <= (Elem x, Elem y);
              //@ behavior {
              //@ ensures returns ⋀ result = (x <= y);
              //@ } };
class PriorityQueue {
public:
//@ uses PriorityQueueTrait(PriorityQueue<Elem> for PQ[Elem]);

          ... constructors have been elided for space reasons ...

virtual void Insert(Elem e) throw();
//@ behavior {
//@ modifies self;
//@ ensures liberally self’ = add(e, selfˆ);
//@ }

virtual Elem Largest() const throw();
//@ behavior {
//@ requires len(selfˆ) >= 1;
//@ ensures result = head(selfˆ);
//@ }

virtual Elem RemoveLargest() throw();
//@ behavior {
//@ requires len(selfˆ) >= 1;
//@ modifies self;
//@ ensures result = head(selfˆ) ⋀ self’ = tail(selfˆ);
//@ }

virtual bool IsEmpty() const throw();
//@ behavior {
//@ ensures result = isEmpty(selfˆ’);
//@ }

virtual long int Length() const throw();
//@ behavior {
//@ ensures liberally result = len(selfˆ’);
//@ }
```

Larch's abstraction mechanism was one of the primary influences for our abstraction research. In the same way that Larch seeks to *express* behavior in terms of abstractions, we seek to use dynamic invariant detection to *discover* behavior in terms of abstractions. In the next section, we look at Java Modeling Language, which incorporates some of Larch's abstraction features as an optional way to specify behavior.

2.1.4. Behavioral Subtyping

Liskov and Wing [Liskov and Wing, 1994], and subsequently, Leavens [Leavens, 2006] refined the idea of behavioral subtyping and addressed some of the problems with Eiffel's handling of argument covariance. JML is the product of this research, and facilitates the expression of specifications that honor their notion of behavioral subtyping.

The idea behind behavioral subtyping is to enforce subtyping relationships that honor the principle of *supertype abstraction* (from [Leavens and Naumann, 2006]):

> The basic idea of modular reasoning, which we call supertype abstraction, is clear. It is a generalization of typechecking: reasoning about an invocation, say E.m(), is based on the specification associated with the static type of E, and constraints are imposed on implementations of m() at all subtypes.
>
> ...
>
> This kind of reasoning, supertype abstraction, is modular in that it does not depend on E's dynamic type, and hence does not have to be changed when subtypes of T [the static type of E] are changed in compatible ways or are added to a program. Supertype abstraction supports maintenance and evolutionary programming styles.

One of the contributions of the work on behavioral subtyping is the idea of the *class invariant* and *history constraint*, which are invariant clauses that specify common behavior for all methods of a class and its subclasses. Such invariants are important for ensuring that the principle of supertype abstraction is enforced. As we described in 2.1.2. Eiffel and Design By Contract, the class invariant is conjoined with the precondition and postcondition of every method. The history constraint (if any), is conjoined with the postcondition of every method, and is intended to specify what a method must do or cannot do. For example, a PatientVisitLog might have a history constraint to restrict methods from modifying or deleting any entries.

In Table 1.1, we described the specification for the ToyMath module as a table where each row corresponds to a Hoare triple consisting of (precondition,method,postcondition). We extend this tabular notation to encompass state variables and the notions of class invariant and history constraint, which we will illustrate using a simple Stack class that contains a single state variable, elements, and the methods push, pop, and reset (Table 2.1). The class invariant in this example is elements != null, which holds in any *publicly observable state*. This particular class has no history constraint, but such a constraint would specify a postcondition that must hold for each method of the class in addition to any per-method postconditions. In 4.3.1. Aggregate Abstraction and History Constraints, we use Alembic to infer potential history constraints.

Table 2.1. A tabular specification of a Stack class that displays the preconditions, postconditions, class invariant, and history constraint. This specification is incomplete in that it doesn't require that push and pop maintain any elements other than the most recently pushed; such specification would require use of loops or universal quantifiers, which is provided by many specification languages.

State Vars	**Class Invariant**	**History Constraint**
Object[] elements	elements != null	*none*
Precondition	**Method**	**Postcondition**
none	void reset()	elements[] == []
none	void push(Object x)	size(elements) == orig(size(elements))+1 x == elements[size(elements)-1]
size(elements) $>=$ 1	Object pop()	size(elements) == orig(size(elements))-1 return == orig(elements[size(elements)])

Another reason for understanding behavioral subtyping is that it was used as the basis for a dynamic invariant detection application that inspired our work on abstraction [Csallner and Smaragdakis, 2006]. In this application, Daikon was used to detect invariants on objects representing the supertype abstraction. This was achieved by synthesizing traces from subtype invocations and copying them to the supertype's invocation. The net result is that Daikon would *see* an amalgam of invocations from the various subtypes, but that they would be viewed through a supertype lens.

In our research on abstraction, we have generalized the notion of propagating traces to supertype objects into a system for synthesizing or transforming traces prior to propagating them to synthesized abstraction objects. In both cases, however, we use a traditional invariant detector to interpret the transformed traces through their projection onto an abstraction object. This will be discussed in detail in Chapter III.

2.1.5. Java Modeling Language (JML)

The Java Modeling Language (JML) is a BISL that is typically used to annotate Java class and interface definitions with behavioral specifications in the form of require and ensure clauses, as well as several other behavior modeling features [Leavens, 2006]. We will be presenting some of the features of JML that are related to abstraction and Alembic, including JML's ghost and model keywords.

Like Larch/C++, JML is primarily used as an annotation language layered upon Java, although JML specifications can also be generated and manipulated separate from any Java implementation class or source file. We provide an example of our ToyMath class in Listing 2.5 below.

Listing 2.5. JML (Java Modeling Language) annotating a Java implementation of theToyMath class (from Leavens and Cheon [2006]).

```
public class ToyMath
{
    public final static double epsilon = 0.0001;

    //@ ensures JMLDouble.approximatelyEqualTo(\result, x * x, epsilon);
    public static double sqr( double x )
    {
        return x * x;
    }

    //@ requires x >= 0.0;
    //@ ensures JMLDouble.approximatelyEqualTo(x, \result * \result, epsilon);
    public static double sqrt( double x)
    {
        return Math.sqrt( x );
    }

    //@ ensures \result <= x
    //@ ensures x - \result < 1
    public static int floor( double x )
    {
        return Math.floor( x );
    }
}
```

There is a relationship between a model field of a JML specification and a feature of a Larch trait. In both cases, the feature exists only in the abstraction, although its value is derived from the concrete implementation. JML differs from Larch in that it embeds this abstraction capability into the interface definition syntax, rather than having a separate shared language like Larch.

2.1.6. Other Specification Systems

Although we have highlighted a small selection of representative specification languages and facilities, there are several others worth mentioning in this dissertation. While these didn't have a direct influence on our research, they are related because they may represent additional application opportunities for Alembic (e.g., as alternate input and output formats), or they may provide an alternate perspective on invariant detection and abstraction. We highlight some of these below.

Z Specification Language This language is similar to the Larch Shared Language in that it is designed for expressing models of software systems [Spivey, 1989]. The emphasis in Z is on using the language to specify models and their properties and behavior directly. This is in contrast to Larch, whose Shared Language expresses relations and facts, but without an underlying data. Instead, Larch provides an interface language to bind the algebraic language to an data structure.

Alloy Alloy [Jackson, 2002] is a specification language based on Z, but with enhanced object modeling facilities and designed with an eye towards automatic analysis of specifications.

2.2. Static Analysis

The previous section introduced the class of behavioral interface specification languages characterized by logical clauses describing preconditions and postconditions. The goal of dynamic invariant detection is to infer these specifications from observing executions. Before we discuss the dynamic inference of these specifications in detail, we will be reviewing at some of the uses and limitations of related static and dynamic analysis techniques. In this section, we will focus on the static analysis of modules.

A static analysis is a technique for examining source code (or something derivable from source code without requiring prior execution) and determining properties of the code. These properties can include statements about whether an exception is thrown, what the result type might be, or whether there are any loops. Due to inherent properties of computation (see Halting Problem below), many static analyses are able to detect and report that a program has a property, but they cannot guarantee that it doesn't have it.

Some static analysis techniques border on the dynamic because they involve simulation, interpretation or evaluation of some parts of a program. In effect, the program is executing dynamically in the mind of the analyst or the memory of the tools. *Abstract interpretation* and *symbolic execution* are examples of this notion, but any programmer who has stepped through a program's source code to understand it is usually performing a form of symbolic execution.

Ideally, we'd like to be able to determine important properties of a program and its modules simply by examining the source code. These properties include:

- Will this program ever crash? For which inputs?
- What are the preconditions and postconditions that a particular function expects?
- How will a particular function's performance scale as a function of its arguments?
- Will this program always halt? If not, for which inputs?
- Can a static analyzer determine that the sum of the first N odd numbers is N^2?

2.2.1. Halting Problem

There are important and fundamental limits to the program properties that can be inferred using software analysis. Many of these limitations occur when we try to use one program (e.g., a software analysis tool) to *understand* a target program in terms of the partial function it implements, where such understanding is usually phrased as, "Does the function implemented by this program have non-trivial property X?". It has been proven that there must be programs for which this question cannot be answered in a general way. These questions can all be rephrased as a *halting problem*, which is the question, "Will this program halt?".

Rice's Theorem [Rice, 1953] generalizes this halting problem by stating that if we have a program written in a Turing complete programming language (i.e., almost any programming language), then there is no general decision procedure to determine whether the function implemented by the program satisfies any particular non-trivial property we might choose to examine. Because a software analysis algorithm is a *general decision procedure*, Rice's theorem means that we cannot write an analyzer that can determine non-trivial properties of programs under analysis for any arbitrary program.

The relevance of this to invariant detection can be understood by recalling how the Hoare specification of a program corresponds to a set-theoretic function between sets of satisfying tuples. An invariant detector (static or dynamic) that could determine the invariants of an implementation would effectively be determining the properties of the underlying function, which is prohibited by Rice's Theorem; therefore, invariant detection is undecidable in general. In practice, however, there are *some* properties that are determinable for *most* programs, and static analyses seek to exploit these for optimization and correctness. But generalized static invariant detection is tantamount to contradicting Rice's Theorem.

One example that illustrates the limitations of static analysis is to consider two implementations of the sqr(float n) function. The first implementation is the obvious one and simply returns n * n. The second implementation computes sqr(float n) in a non-obvious fashion. For example, we might sum the first n odd numbers to achieve sqr; or we might add n to itself n times. Static invariant detection would allow us to determine that these implementations both have the postcondition invariant of return == n * n; however, such a static analyzer would have to understand general properties of natural numbers in order for it to make this judgment. This cannot be done in general, because it would require a decision procedure for arbitrary statements about the natural numbers, which is impossible as shown by Gödel's Incompleteness Theorem.

Unlike static invariant detection, dynamic invariant detection is able to discover the *observation invariants* for both implementations of sqr above, and these observation invariants imply (or are identical to) the invariants that describe the behavior of sqr. Indeed, one of the promising features of dynamic invariant detection is its ability to find these program properties even if the implementation of the program is complex or otherwise inscrutable. One caveat with the observation invariants reported by dynamic invariant detection is that they only hold for past observations; it is very possible that the observations do not include a crucial input that would cause the program to crash or loop forever, meaning that observation invariants cannot be treated as *proven* or *true*.

2.2.2. Symbolic Execution

Symbolic Execution [King, 1976] is a static analysis technique that produces symbolic descriptions of procedures in terms of the effect of the procedure on state variables and results as a function of the input state variables and arguments. This is done by simulating the execution of the procedure, but instead of variables storing actual values, they will store symbolic expressions during this simulation. Branch points are encoded into these expressions so that the resulting expressions will describe any conditional behavior. In the absence of loops or recursion, symbolic execution is able to exactly characterize the internal behavior of a procedure as a set of expressions.

Symbolic execution is based upon exploring *all possible worlds*, which becomes problematic in the case of loops or recursion, because the analyzer can get into a loop. Dynamic Symbolic Execution is a dynamic analysis technique that performs similar reasoning to the static version, except that instead of all possible worlds, only those paths actually dynamically executed will inform the symbolic execution [Csallner et al., 2008]. We discuss this more in 2.5. Dynamic Symbolic Execution (DySy).

2.2.3. Abstract Interpretation

The static analysis technique known as *abstract interpretation* has significant parallels with our work. The principle of abstract interpretation is to determine an abstract semantics or specification for a program fragment, and to apply traditional static analyses to this abstraction [Cousot and Cousot, 2004] [Schmidt, 1998]. Contrast this with Alembic, which determines an abstract program and uses a traditional dynamic invariant detector upon the abstracted program.

During the formation of Alembic abstractions, it is often helpful to use the principles of abstract interpretation. For example, if we are performing an *Effect Abstraction*, capturing the CPU time metric cost of targeted function calls, we might wish to discretize the CPU time to make it more amenable to Daikon analysis. We believe that the rich catalog of abstractions and techniques developed for abstract interpretation can be a useful source of abstraction patterns for use with Alembic.

2.2.4. Model Checking

Model checking is a static analysis technique based upon abstracting a concrete program into a *checkable model*, and then using algorithms to evaluate safety and liveness properties of the model. The model is an abstraction of the concrete program, usually containing fewer and more discrete states than the concrete program. Often, the model is developed first in a specification language such as Alloy, and then evaluated with a model checker such as the Alloy Analyzer before ultimately being implemented in code. A model checker typically seeks to discover if a given predicate could ever be satisfied, and if so, what values of variables would satisfy it. If this predicate is something like *will this plane ever get into an uncontrolled stall?*, and a suitable model of the flight control system were specified, then a model checker could report the conditions under which the predicate is satisfied.

2.3. Dynamic Analysis

One of the advantages of static analysis over dynamic analysis is that it can be used to check important properties during the development process, informing the programmer about potential problems or non-conformance with a specification or requirement. At a minimum, static analyses are a key component to compiler optimizations and bug detection mechanisms. Dynamic analysis techniques are most applicable where we require knowledge of transient values and metrics, such as profiling or assertion-checking; or, as we showed above in Static Analysis, where we seek properties denied us by Rice's Theorem.

An analogue of the static versus dynamic analysis distinction can be found in applied mathematics, where there are many problems which cannot be solved symbolically as closed-form expressions (static analysis), but are amenable to numerical methods when the problem is bound to actual values and parameters (dynamic analysis).

We can partition dynamic analyses into those that operate at the *syntax* level of the executing program and those that operate at the *implementation* level. A syntax-based dynamic analysis measures properties associated with source code elements such as functions, variables, arguments and results, in addition to intra-procedural elements such as loops, conditionals and assignments. These analyses include dynamic invariant detection, dynamic symbolic execution, bounds and pointer checking, and assertion checking.

The other type of analysis measures properties associated with implementation artifacts such as number of instructions, execution time and space, stack depth, and similar runtime properties. Examples include profiling of resource usage (time and space, minimally), coverage analysis, and concurrency analysis. Even though dynamic invariant detection is a syntax-based analysis, we can use Alembic to capture some of these implementation properties and *lift* them into traits where they can be subject to dynamic invariant detection. We call this technique *Effect Abstraction* and describe it briefly in 5.1.4. Other Ideas.

Two common forms of dynamic analysis are profiling and coverage analysis. Profiling associates resource metrics (space or time) with various *events* that occur during execution; minimally, these events include the entry and exit of a function. Profilers may obtain these metrics either by *transforming* the program to capture and emit the measurements, or by externally *sampling* the metrics via operating system facilities. Another useful dynamic analysis is bounds and pointer analysis, which can assist in debugging a program or detecting misuse of the heap or a data structure by revealing leaks, dangling pointers, and array bound violations.

There are three primary ways to obtain the necessary information for performing a dynamic analysis:

Instrumentation The program may be transformed into an instrumented program by inserting code at various observation points (typically function entry and exit). This code then captures and emits the data needed for analysis. Instrumentation often affects the performance of the instrumented program negatively.

Sampling The operating system or other mechanism periodically takes a snapshot of the program's address space and uses data from this snapshot to perform dynamic analysis. Sampling is ideal when the target program cannot be instrumented, but it is imperfect in that not all events will necessarily be observed, depending on the sample frequency.

Simulation While this is properly a static technique, it involves running the program in some virtual machine that provides all of the needed observational data to the dynamic analyzer.

Both Daikon and Alembic rely upon instrumentation of the target program to capture and emit the data required for invariant detection.

2.4. Dynamic Invariant Detection

Although there are several important optimizations in the Daikon detector, the fundamental idea is easily understood [Perkins and Ernst, 2004]. At initialization, Daikon prepares a set of *candidate invariants* for every module targeted for analysis. These logical expressions are generated from a lexicon of variable names and operators that are visible and appropriate for each public function entry and exit declaration. As the program executes, traces are emitted to the detector for each targeted function entry and exit; these traces contain the values of arguments, state variables and results. For each trace, Daikon examines its set of candidate invariants and eliminates from the set any that are falsified by the observed trace data. Whatever candidates remain at the end of a suitably robust execution are *presumed true* and can be considered as potential specifications. At the very least, the remaining invariants are consistent with the observed executions, which may indicate bias or weakness in the execution framework.

A dynamic invariant detector can be viewed as a process that takes as input a program (in source or binary form) and one or more valid executions of that program, and produces as output an *operational abstraction* for each targeted module (e.g., Java class) within the program. The operational abstraction summarizes the observed behavior of a module in terms of properties that are externally visible, such as input arguments, state variables and return values. This behavior is reported as as set of structural invariants and function precondition/postcondition pairs[1].

The Daikon [Ernst et al., 2007] dynamic invariant detector accomplishes the above by using trace information from an instrumented program as input to an inference engine that generates a set of candidate invariants and then eliminates those not supported by the trace observations. The resulting invariants associate logical propositions with objects or modules (*structural invariants*), and with function entry and exit points (*precondition/ postcondition*).

In the dynamic invariant detection terminology, a *program point* refers to a point in the program text at which traces are collected and from which inferences are made. Each program point has a set of visible variables such as input arguments, results, and object state variables.

In addition to the program points corresponding to function entry and exits, Daikon also considers the synthetic CLASS and OBJECT program points, where we can measure the structural invariants of an object, class or module that hold outside of a function execution. Figure 2.3 illustrates the important program points for an example class Foo and a method bar within the class.

2.4.1. An Example of Dynamic Invariant Inference

We present a simple example of Daikon usage, from the initial instrumentation of a target program to the execution. Assume that we have a Java class, Simple, that contains a single instance method, .m() and no instance variables as in Listing 2.6. We exercise this class with a main program as in Listing 2.7.

We use the script in Listing 2.8 to compile, instrument and execute the program. The —ppt-select-pattern is present as an optimization to ensure that Daikon only needs analyze the Simple class and its implementation of .m(). The default behavior of Chicory is to instrument the class files and then to execute the main class given on the command line (SimpleTester, in the above example). The traces generated as each instrumented function is entered and exited are typically output to a trace file, which is then read by Daikon for subsequent invariant inference.

We use the script in Listing 2.9 to invoke the Daikon detector upon the trace file generated by Chicory above.

[1]The terms *operational abstraction* and *invariant* will be used interchangeably in this document following usage in the invariant detection literature. A more accurate terminology is that an operational abstraction is a specification in terms of invariants and assume/guarantee constraints.

Listing 2.6. Java code implementing a simple class named Simple with a method .m().

```
public class Simple
{
  public Simple() // Object constructor
  {
  }

  public int m( int input ) // Invariants will be observed on this method
  {
    return input ∗ input;
  }
}
```

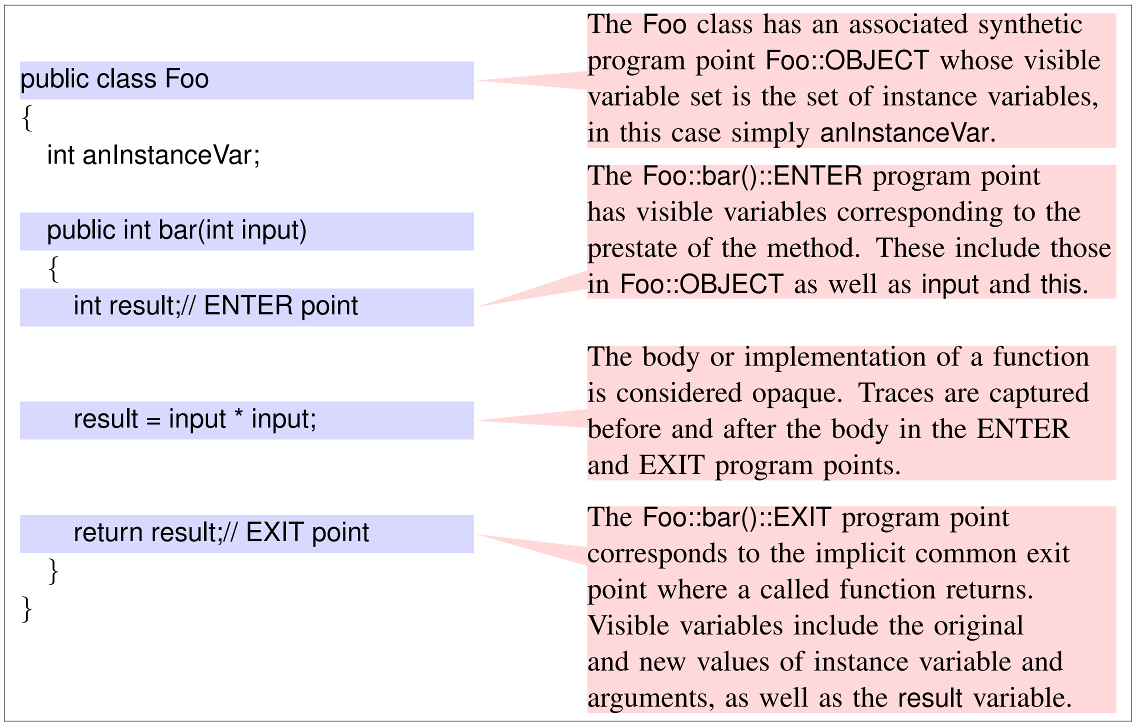
Listing 2.7. Java code exercising Simple.

```
public class SimpleTester
{
  public static void main(String[] args)
  {
    Simple simple = new Simple();

    for ( int i = −100; i <= 100; ++i )
    {
      assert simple.m( i ) == i ∗ i; // Give method .m() data for inference
    }
  }
}
```

Listing 2.8. A shell script to compile, instrument and execute the SimpleTester program with Chicory.

```
javac −g ∗.java ## Compile
java \ ## Instrument and Execute
    daikon.Chicory \
        %\dd%dtrace−file=SimpleTester.dtrace.gz \
        %\dd%ppt−select−pattern='Simple.m|Simple:::OBJECT|Simple:::CLASS' \
    SimpleTester
```


```
public class Foo
{
  int anInstanceVar;

  public int bar(int input)
  {
    int result;// ENTER point


    result = input * input;


    return result;// EXIT point
  }
}
```
The Foo class has an associated synthetic program point Foo::OBJECT whose visible variable set is the set of instance variables, in this case simply anInstanceVar.
The Foo::bar()::ENTER program point has visible variables corresponding to the prestate of the method. These include those in Foo::OBJECT as well as input and this.
The body or implementation of a function is considered opaque. Traces are captured before and after the body in the ENTER and EXIT program points.
The Foo::bar()::EXIT program point corresponds to the implicit common exit point where a called function returns. Visible variables include the original and new values of instance variable and arguments, as well as the result variable.

Figure 2.3. A function and its associated program points and visible variables. For each program point, there is an associated set of visible variables. Execution that proceeds through a program point will obtain the values associated with that point's visible variables. This is called a *trace*.

After executing the instrumented program and analyzing the traces, Daikon prints the invariants displayed in Listing 2.10. Note that we have slightly reformatted Daikon's output for readability; also, we only present the entry/exit invariants for method .m().

The invariants printed by Daikon in Listing 2.10 reveal the following potential invariants:

- There is no discernible invariant associated with the input variable at Simple.m(int):::ENTER. One would expect our invariant detector to discover that input >= -100 and input <= 100, but the Daikon detector does not discover this

Listing 2.9. A shell script to analyze the generated tracefile using Daikon.

```
java \ ## Analyze
    daikon.Daikon \
        %\dd%config_option daikon.Daikon.progress_delay=−1 \
        %\dd%format JML \
    SimpleTester.dtrace.gz
```

by default; it requires command-line options to enable the detection of this type of bounds.

- Daikon discovers two potential invariants associated with Simple.m(int):::EXIT. The first is our desired invariant, return == orig(input)**2, which indicates that the return value is the square of the input value. The other invariant, return >= orig(input), is also correct, but not as useful because it is implied (by the laws of arithmetic) by the first invariant.

2.4.2. Uses for Invariant Detection

There are a variety of applications of traditional dynamic invariant detection, and our research enables some new ones. Described below are some of the common uses:

Understanding The invariants reported can be used to guide understanding of a module, even when source code is unavailable or obtuse. This understanding can be used to confirm or disprove assumptions and expectations about a module, revealing bugs or poor documentation.

Specifications The invariants discovered by dynamic invariant detection can be used to automatically build specifications for modules. These specifications can then be used by tools such as JML and ESC/Java2 to verify that the implementation conforms to the specification, or to enforce the invariants at runtime [Nimmer and Ernst, 2002]. It has been observed [Csallner and Smaragdakis, 2006] that simply taking the invariants produced over method implementations is not sufficient to build specifications that reflect behavioral subtyping. Our research was originally motivated by a desire to enhance the detection mechanism to address this problem. We believe that the abstraction techniques can solve this as well as other unanticipated problems.

Test Coverage Reported invariants not only reveal the implementation of the targeted modules, but also some aspects of the test framework that exercises the

Listing 2.10. Daikon-inferred invariants for class Simple upon method .m().

```
Simple.m(int):::ENTER
    (no invariants detected)

Simple.m(int):::EXIT
    return == orig(input)**2
    return >= orig(input)
```

modules. For example, if the discovered precondition for method foo(int n) is ($n > 0$), then this might reveal a weakness in our test framework; perhaps we should be exercising foo with negative and zero inputs.

2.4.3. How Daikon Works

We will illustrate how an incremental dynamic invariant detector works by describing how Daikon performs its inference on the above example. The original version of Daikon used a simple algorithm that we describe below. Subsequent optimizations increased the performance and scalability of Daikon [Perkins and Ernst, 2004], but do not substantially alter the description for our purposes here.

The basic steps in using Daikon are to compile and instrument the target code with the Chicory tool, to execute the code, and to analyze the resulting traces with the Daikon analyzer; this process will output a set of invariants describing the observed behavior. See Figure 2.4 for an overview of the process, which we detail in the rest of this section.

2.4.3.1. Instrument the Program with Chicory

The first step is to mechanically instrument the compiled Java class files. In the Daikon system, this is performed with the Chicory tool, which typically instruments and then executes the code. Chicory reads the class files and inserts tracing code at each implementation function's entry and exit points[2]. These entry and exit points are examples of *Program Points* in Daikon's model.

During the execution of an instrumented program, whenever control passes through an instrumented program point, a trace tuple is emitted which records the values of arguments and member variables that are visible at that program point. For example, in the case of the exit program point Simple.m(int):::EXIT, the visible variables are simply this, input and return, where return indicates the result value that will be returned to the caller of .m(). If class Simple had data members, then their values would also be visible at this program point.

It should be noted that Daikon's Java tracing does not include inherited data members, even if they are protected or public. The Turnip system of Kuzmina and Gamboa addresses this potential problem [Kuzmina and Gamboa, 2007] as is discussed in 2.6.3. Polymorphic Analysis.

[2]Chicory uses the Byte Code Engineering Library (BCEL) to analyze Java class files and to insert the appropriate instrumentation. A version of BCEL is also incorporated into AspectJ for its code-weaving purposes.

2.4.3.2. Execute the Program

Once the compiled code has been instrumented with Chicory, it is executed under the direction of either a test program intended to exercise the target code, or it is executed as part of a larger system (not all of which need be instrumented). The traces generated as each instrumented function is entered and exited are typically output to a trace file, which is then read by Daikon for subsequent invariant inference.

The important part of execution is to fully exercise the targeted modules to provide Daikon with as much information about the program points' data values as is practical.

2.4.3.3. Initializing Daikon

One of the artifacts produced by Chicory as it instruments the code is a declaration file that contains description of the program points discovered as well as the variables that are associated with these program points. This file is read by the Daikon analyzer prior to reading any data traces; in fact, it is usually just prepended to the trace file.

Daikon reads in the declaration file and builds a set of *candidate invariants* that might apply to the loaded program points. The candidate invariants for a given program point (e.g., the Simple.m(int):::EXIT point) are constructed from a set of predefined templates of possible relational expressions, which are then instantiated over the variables visible at that program point. The templates available in Daikon include unary, binary, and ternary relations and include common relations such as arithmetic $<$, $>$, and ==, as well as more sophisticated relations such as $x \in$ someList.

When instantiating templates, Daikon uses both the primitive variables visible at a program point, as well as *derived variables* that are built by combining or transforming the primitive variables. For example, if aList[] is a Java array variable visible at a program point, then Daikon would also consider the derive variable size(aList[]), which represents the length of the array. Daikon has a configuration option (disabled by default), that would additionally consider aList[0], aList[1], and aList[-1], which are the first, second, and last array element, respectively. If the variable idx and aList are visible at a program point, then Daikon will consider invariants that involve the derived variable aList[idx].

2.4.3.4. Elimination via Falsification

As a result of the combination of its rich set of invariant templates and derived variables, Daikon is able to construct a large set of candidate invariants. After initialization, Daikon will begin reading the data traces produced by the instrumented program. A data tuple contains the values of visible variables at each instrumented function's entry and exit point.

For each data tuple, Daikon examines the set of candidate invariants and eliminates those that are falsified by the evidence in the data tuple. As a consequence, the bulk of the candidate invariants are eliminated rapidly (after a few tuples), and the resulting set of candidate invariants are those that have not yet been falsified via evidence. If the unit test or framework executing the target code is sufficiently rich and exercises the spectrum of possible legal calls to the target, then the invariants that remain unfalsified can be treated as approximate behavioral descriptions.

Daikon uses a parameter known as the *confidence limit* as the basis of a statistical test that will suppress invariants that haven't seen sufficient evidence. The term *likely invariant* is used to refer to those unfalsified invariants that have met the desired confidence limit. It may be that the reported invariants appear true because of weaknesses in the coverage of the exercising program; enhancing the test suite to cover more cases may falsify these invariants. A software engineer analyzing partial results may still be able to derive useful behavioral descriptions or determine that the test suite is insufficient.

2.4.3.5. Limitations of this Approach

Because dynamic invariant detection relies upon evidence-based inference, the accuracy of the detected invariants will be dependent upon the completeness and coverage of the unit test or program that exercises the modules under test. A gap in the testing framework will result in unfalsified invariant candidates that will be reported as potential invariants. However, this characteristic of dynamic invariant detection can be used as a feature when trying to construct exhaustive tests. Incorrect reported invariants can reveal incompleteness in the testing regimen.

Another limitation of the template-based approach is that the set of templates is restricted in size and scope, in order to perform reasonably in the general case. As we increase the depth of expressions considered, the number of possible operators, the number of invariant types, and the number of visible variables, we also increase dramatically the number of candidate invariant to be processed.

Alembic is designed to address this by augmenting the template-based implicit abstractions with explicit abstractions that may express arbitrarily complex expressions. An alternative solution to the limits of templates is discussed in 2.5. Dynamic Symbolic Execution (DySy).

2.4.4. Alternative Schemes

Daikon's contribution to dynamic invariant detection is twofold. First, Daikon provides a *front-end* for instrumenting a target program to emit traces containing entry and exit values and expressions. Second, Daikon provides a *back-end* to analyze traces and infer specifications. As we will show in Chapter III, Alembic provides an alternative front-end that uses AspectJ to instrument the program.

Although it is not in the scope of this dissertation, we have considered alternative back-end schemes for deriving invariants from trace data. Alembic was designed to allow the potential use of such alternate back-ends. The fundamental back-end problem of dynamic invariant detection is about finding a model that explains a set of observed data. This model is to be expressed as a set of invariants (preconditions and postconditions).

Some of the approaches that can be used:

- Search
- Genetic Programming
- Grammatical Evolution
- Numerical Regression

2.5. Dynamic Symbolic Execution (DySy)

One of the potential drawbacks of the Daikon technique for discovering invariants is that the grammar of the invariants detected depend upon the set of templates and derived variable forms that are built into the detector. Ideally, the most appropriate specification of a module under analysis can be discovered by combining the appropriate templates and derived variables. However, in order to achieve acceptable performance, Daikon does not consider all possible invariants, but only the finite set derivable from its finite list of templates and derived variables.

This means that the invariants generated may not be the most natural or useful ones. This shows up most clearly with conditional invariants. A conditional invariant is a set of cases, where each case consists of a precondition and a postcondition. Conditional invariants naturally express conditional behavior of functions. However, a tool such as Daikon does not derive conditional invariants except when given explicit directives regarding the *splitting condition* that is used to build the precondition expression, or when the function is boolean or has multiple exit points.

The DySy (Dynamic Symbolic Execution) system [Csallner et al., 2008] addresses this problem by using the principles of dynamic invariant detection upon a symbolic execution engine. DySy generates expressions that are not derived from an arbitrary set of templates, but that are derived from the dynamically observable symbolic expressions present in the executing code.

2.6. Related Work

The research described below was influential in the development of abstraction and Alembic. We briefly describe the relevant aspects of each.

2.6.1. Contract Soundness

Findler and Felleisen developed a specification and checking framework known as Contract Java [Findler and Felleisen, 2001] that supports the creation of programs that embed contracts as part of the language definition. These contracts are checked at runtime for violation of preconditions and postconditions, as well as for checking for behavioral subtyping hierarchy errors. The work is significant for our purposes here because the mechanism used by Contract Java for runtime checking of hierarchy errors is similar to that we will be using to propagate data traces.

2.6.2. Inferring Behavioral Subtypes

The motivating example for this research was described by Csallner and Smaragdakis [Csallner and Smaragdakis, 2006], where they outlined problems with Daikon's specifications. Specifically, they presented an example where the JML annotations produced by Daikon are inconsistent with behavioral subtyping, resulting in a failure in the ESC/Java2 (Extended Static Checker for Java) tool [Cok and Kiniry, 2004].

The key idea extracted from this paper was the idea of *trace propagation*, where concrete executions of methods upon dynamic receivers (subclass instances) were reflected as though they were also executions upon the superclasses that also contained those methods. This is a form of *aggregate abstraction*, where traces from multiple program points are joined to produce invariants that summarize the behavior of the component program points.

We realized that the notion of trace propagation and the associated notion of trace transformation could be applied to a variety of problems. We also believe that the solution in the paper above has some drawbacks. Specifically, by feeding subclass traces to the superclass method, we dilute the information obtainable by an invariant detector because we can no longer get invariants over the concrete superclass implementation. Instead, the superclass's implementation of a method becomes the target for observations from its subclasses. In addition, the reliance upon a concrete superclass means that abstract superclasses or interfaces cannot be used as targets.

We address this with Alembic by forcing the definition of a parallel class hierarchy that will contain the aggregate observations. This allows us to use our detector on both hierarchies: one reveals implementation details about each concrete class, the other reveals aggregate behavior corresponding to behavioral subtyping. This approach is described briefly in 5.1.2. Supertype Abstraction.

2.6.3. Polymorphic Analysis

Kuzmina and Gamboa have created a dynamic invariant detector similar to Daikon that they call Turnip [Kuzmina and Gamboa, 2007]. It uses a method similar to Daikon's. They have developed improvements to obtain behavior specifications that are broken into cases, based upon the dynamic receiver of a method. In effect, the invariants are conditional invariants where the preconditions for the cases are of the form `this.getClass() == SomeClass.getClass()`. They do this by augmenting the instrumentation layer to include the dynamic class of the receiver, as well as those member variables that are visible from that class. The resulting case-based invariants are sometimes more informative and useful than Daikon-produced invariants.

2.6.4. Variable Hierarchy

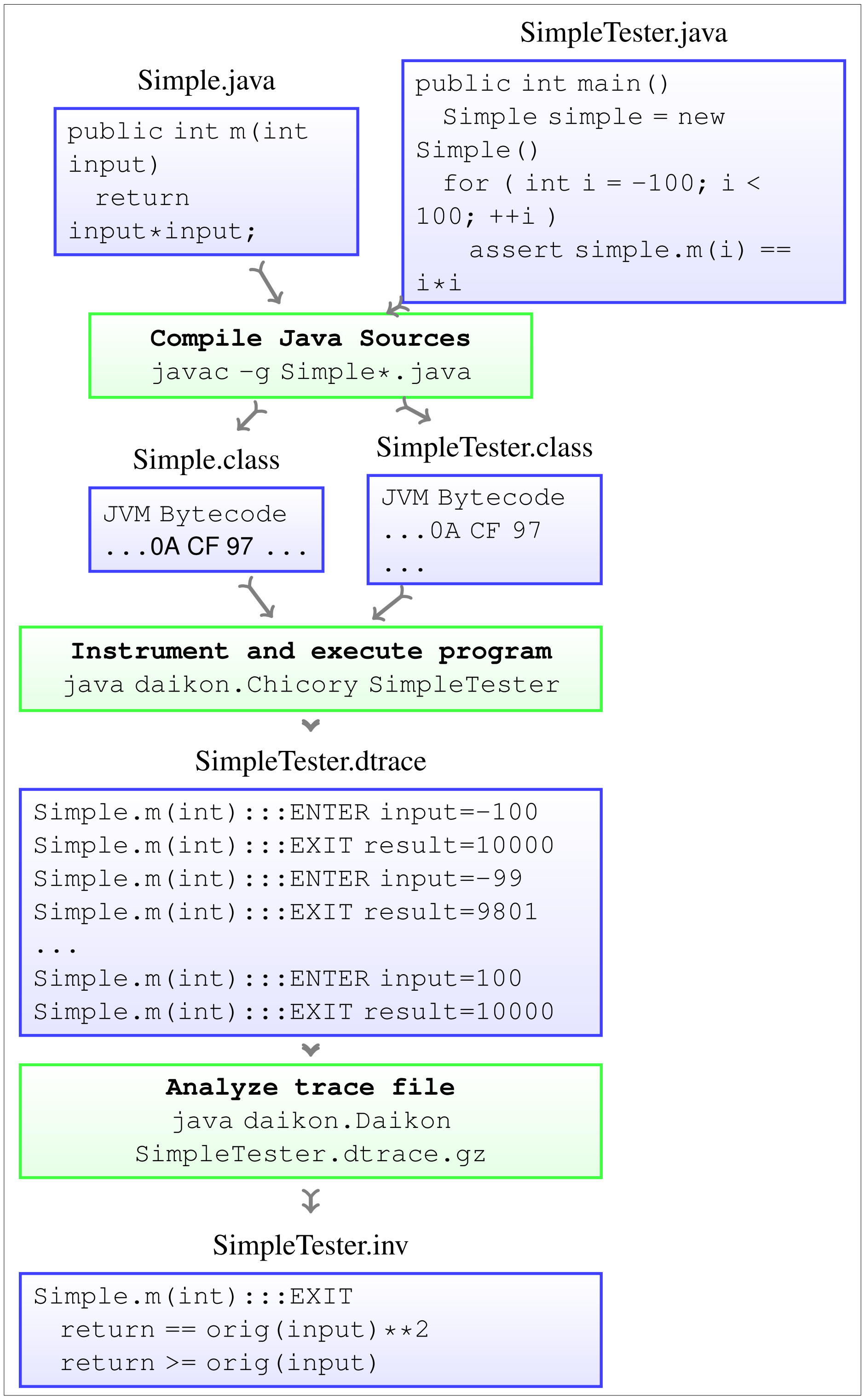
Nimmer and Ernst considered several optimizations and extensions to the original Daikon implementation, several of which were eventually incorporated. The idea of propagating data traces to superclasses and to abstract interfaces was suggested, but never implemented [Nimmer and Ernst, 2002].

Daikon does use some of the features suggested by Nimmer in the form of the *variable hierarchy* optimization. This feature allows data traces from child program points to be copied or *propagated* to parent program points. The two main uses for this feature in Daikon are:

- Traces from internal function exits are propagated to the terminal function exit, which summarizes the aggregate behavior of all exits from the function.
- The pre- and post- values of state variables for each instrumented function invocation are propagated into the object invariant, which summarizes the aggregate invariant common to all entries and exits in the class.

The abstraction techniques described in this dissertation are similar to the variable hierarchy propagation in Daikon; the aggregate abstraction techniques we describe in 4.3.1.

Aggregate Abstraction and History Constraints rely upon the same principle of propagating traces upwards to an aggregation abstraction. However, the hierarchy in Daikon derives from the program text, and there is no transformation of values between the child and parent in the hierarchy. Alembic allows transformation of values, selective propagation, and the synthesis of arbitrary abstractions.



```
public int m(int
input)
  return
input*input;
```

```
public int main()
  Simple simple = new
Simple()
  for ( int i = -100; i <
100; ++i )
    assert simple.m(i) ==
i*i
```

```
javac -g Simple*.java
```

```
JVM Bytecode
...0A CF 97 ...
```

```
JVM Bytecode
...0A CF 97
...
```

```
java daikon.Chicory SimpleTester
```

```
Simple.m(int):::ENTER input=-100
Simple.m(int):::EXIT result=10000
Simple.m(int):::ENTER input=-99
Simple.m(int):::EXIT result=9801
...
Simple.m(int):::ENTER input=100
Simple.m(int):::EXIT result=10000
```

```
java daikon.Daikon
SimpleTester.dtrace.gz
```

```
Simple.m(int):::EXIT
  return == orig(input)**2
  return >= orig(input)
```

Figure 2.4. The dynamic invariant detection process and intermediate files involved in instrumenting, executing, and analyzing the Simple example.

CHAPTER III

ABSTRACTION AND ALEMBIC

The central questions of our research are: *Can dynamic invariant detection be applied to user-defined abstractions of concrete modules?*, *Is this useful?*, and *Is it practical?*. In this chapter and the next, we answer these questions positively by showing several ways to use abstraction techniques with a dynamic invariant detector, and by showing that the results obtainable add value to the basic dynamic invariant detector.

We begin this chapter by reviewing the abstraction mechanisms available to dynamic invariant detectors; these include source code modification and the use of simple abstraction facilities such as Daikon's purity-file. We then consider the possibilities afforded by creating synthetic traces to abstraction classes, and how these can be injected into our invariant detector, causing it to perform inference upon a model program, rather than a concrete program. We illustrate this by showing how a crude abstraction mechanism can be built upon Daikon by hand-coding the instrumentation code and the abstraction class.

We then introduce Alembic as a system that can automate the above synthesis and abstraction process, and as a declarative language that can easily express the desired abstractions at a high level. Using Alembic, we will construct several example abstractions as a way of explaining Alembic, as well as revealing some of the abstraction possibilities. In the next chapter, we will answer the remaining two questions above about the utility and practicality of abstraction.

In the next chapter, we will also present more sophisticated abstractions that cannot be practically expressed within the context of existing dynamic invariant detectors. However, Alembic enables the creation of explicit abstractions whose behavior can be observed by a dynamic invariant detector to produce useful results on the abstractions. These can then be used to better understand the concrete module.

3.1. The Importance of Abstraction

We begin with a brief reminder of what we hope to gain by abstraction. An invariant detector that is operating upon a concrete module will report invariant expressions built from the concrete state variables, arguments and results visible at a function entry or exit point. A general-purpose detector such as Daikon will explore a finite search space of expressions containing subsets of these variables. Some expressions will *never be considered* due to the necessity of a finite search and the resulting limited depth of expressions explored.

Another reason that some expressions are not considered is that they may involve operations or relations that are not in Daikon's general-purpose vocabulary. In addition, a detector may not be able to *see* important and relevant quantities that are only obtainable via functions or system calls. Finally, even if a true and relevant invariant is discovered, it may be phrased in terms of concrete variables that contain opaque or convoluted data structures, and will therefore remain inscrutable to a human analyst.

Abstraction techniques let us address all of the above by providing a way for a user to guide the search and shape the results. This can be done by building abstractions that ignore irrelevant variables, that make visible the quantities that would ordinarily be hidden, and that create higher-level operations and representations more suited to understanding (human or mechanical). These explicit abstractions do not always require in-depth knowledge of the concrete class's implementation and variables. Often, the raw information needed for abstraction is easily available as one or more public accessor functions on the object. For example, we can use the size() method, rather than traversing an opaque internal structure (provided we trust the size() function is correct).

3.2. Guidelines for Abstraction Use

The proposal that users learn and use an additional language (Alembic) in order to be able to perform refined analysis brings up legitimate questions about whether the effort is worth the reward and how best to discover or invent abstractions. Alembic is designed to allow an analyst to try out several different abstractions simultaneously; this is to encourage experimentation. Alembic ensures that even though multiple views may be associated with a given method, the underlying concrete method will only be invoked once.

The development of abstractions is a balance between several factors:

- The cost of the representation function lift_state impacts the performance of all instrumented methods on a class.
- There is an advantage to using simple datatypes in the abstraction (e.g., a native Java array instead of a linked list) because the invariant detector is limited in its vocabulary.
- Including more variables in the trait means that more invariants will be produced as the detector tries to relate them to each other.
- If state variables are grouped into different traits, each containing related variables derived from the concrete module, then irrelevant invariants are less likely to be produced. Each trait is a filter that reveals one facet of the underlying behavior; each facet may consist of one or more derived state variables.

- A trait need only contain potentially relevant state variables that are meaningful in the abstraction. Concrete variables that are primarily reflective of *what* an implementation does rather than *how* it does it are good candidates for lifting. Often all concrete state variables can be eliminated in favor of a few abstraction variables derived from pure (readonly) instance methods.
- Depending upon the dynamic invariant detector used, there are limitations in the types of relations and operators used in invariant clauses. This can constrain the datatypes that can practically be used in an abstraction. Abstraction can be used to transform data values into types more digestible by a dynamic invariant detector.

Even using a traditional dynamic invariant detector will require some tuning of the configuration options to elicit good results. The effort required to tune Daikon for a particular problem is tantamount to that required to develop simple abstractions that generate good results. In the MMPQ comparison of Daikon and Alembic in Chapter I, the shell script written for Daikon and the MMPQ.alembic file written for Alembic had a similar level of complexity.

3.3. Existing Abstraction Mechanisms

Before we begin our discussion of Alembic in the next section, we will describe some of abstraction techniques available to a Daikon user, although some of these will be adaptable to any dynamic invariant detector.

3.3.1. Implicit Abstraction

We should first observe that a trace-based dynamic invariant detector (e.g., Daikon) is already performing an abstraction on the concrete implementation. This is a *lossy* abstraction, where data is gathered only partially, or not at all.

The traces that are generated at program points with a tool such as Daikon's Chicory are actually abstractions of the program state at those points. Only a subset of the available information is recorded, and thus available for use by an inference engine.

The information typically captured by Daikon includes:

- The value of the dynamic receiver (e.g., this)
- The value of the dynamic receiver's class (e.g., this.class)
- The value of the dynamic receiver's immediate members
- The value of any arguments and their immediate members
- The return value of the function and its immediate members

Information not captured by Daikon's Chicory includes:

- The *deep* structure of the receiver or its arguments. There is a limit to how much nested structure is recorded. The default in Daikon is to record the immediate member values of a variable. If those values are structured, then only a limited depth of this structure is captured in the trace information.
- The static receiver class. This is essential for the propagation of traces necessary to deduce behavioral subtypes, as in the behavioral subtyping example.
- Other aspects of the calling context, including the caller's caller. Nimmer and Ernst [Nimmer, 2002] discuss some of the possibilities available by using this context as a basis for invariant detection.
- Inherited data members of the target object, state variables, and arguments. See 2.6.3. Polymorphic Analysis for an alternative to this limitation.

In addition, the instrumentation mechanism only performs these abstractions over method executions, and not over more complex control structures such as loops and compositions.

Our research provides a mechanism to abstract arbitrary amounts of state and argument data and place it into an abstraction object, where it is made visible to a traditional detector. In addition, we can place the detector's probes anywhere we choose, synthesizing traces corresponding to a variety of control and composition constructors. Some of the diverse applications of these capabilities are described below.

3.3.2. Filtering

One of the best ways to reduce the flood of invariants is to use filters to restrict which methods are examined, and to restrict which variables are considered for use in invariants. This is effectively a *projection abstraction*, where we are projecting from the full set of concrete methods and variables into an abstraction with a potentially smaller set of both. Daikon provides the ability to control the invariants that are reported by filtering method names with the –ppt-omit-pattern and –ppt-select-pattern command options, and by filtering variable names using the –var-select-pattern and –var-omit-pattern options [Ernst, 2010]. Daikon can also restrict the data that is included in the trace stream by use of the –var-omit-pattern option.

For all of our experiments, we use –var-omit-pattern to eliminate the .class variable that is added to all objects, because this variable usually results in a large number of uninteresting invariants.

There is a problem with Daikon if one tries to restrict the variables at a program point too much: member variables cannot be isolated. Excluding the this variable from the invariant lexicon will also exclude all variables of the form this.foo, where foo is a member variable. This means that it is impossible to obtain the simple results we've shown with Alembic (see example in Chapter I) using Daikon. Admittedly, this limitation is not inherent to invariant detectors and may be an addressable problem bug within Daikon.

3.3.3. Inline Abstracted State

Although an invariant detector such as Daikon has a rich vocabulary of invariant templates, many modules have properties that are not visible or describable via the standard templates. One way to address this is to extend an implementation of such a module to maintain an additional set of state variables that collectively represent an abstraction of the original module, and to modify each of the module's called methods to update these derived fields at entry and exit. The invariant detector can then be directed to focus only on invariants that include these derived fields and to exclude invariants referring to the original concrete fields.

Effectively, the desired abstraction has been merged with the concrete object, and the invariant detector has been directed to ignore the bulk of the concrete object. This parallels the ghost field of JML (see Java Modeling Language (JML)), which is a specification-only field that is updated via explicit specification directives.

This technique, while guaranteed to work in any invariant detector, is tedious and requires modification of the source code to implement the updating of the derived abstraction variables at entry and exit. In the next section, we describe a feature of Daikon known as a purity-file that simplifies this process.

3.3.4. Inline Pure Methods

In the case of Daikon, it is possible to declare a set of member functions to be *pure*, meaning that they promise to not modify any object state when called. This capability is enabled via the –purity-file option, which allows functions to be listed in a file. These functions will be evaluated at function entry and exit and the results will be included in the trace stream as though they were ordinary member variables. This enables Daikon to report invariants over a larger lexicon that includes these pure functions.

This mechanism of including the value of pure functions in the trace stream is a form of abstraction, where the pure function is presumably abstracting some values derived from the object's state into a new value that is more meaningful. For example, the size() method for a container class is a good candidate for being a pure function. It abstracts a complex data structure (e.g., a tree) into a much simpler value (e.g., an integer) that is more useful to an analyst and that is more amenable to the limitations of a dynamic invariant detector, which may be limited in the types of data it understands.

This parallels the model field of JML (see Java Modeling Language (JML)), which is a specification-only field whose value is derived from fields in the concrete object. This mechanism is similar to the basic feature of Alembic which enables a trait's state variable to be derived from the concrete object via a lift_state clause. However, the use of inline abstractions and an associated *purity-file* has the following problems:

- There will not always be an existing pure method on the class that expresses a desired abstraction, so an analyst might need to add such a method. However, this violates the separation of concerns by mixing the analyst's abstraction and instrumentation code with the implementor's code.
- Daikon sees no difference between the pure abstraction functions and the concrete state variables, so in its search for invariants, it will report relationships between the pure function and the concrete state; these relationships are true, but irrelevant, and they distract from the important invariants that the analyst seeks. We use the term *cross-talk* for this problem of the detector trying to infer relationships between the concrete and the pure function. It is a general problem whenever an abstraction is placed inline with the thing that is being abstracted. Alembic addresses this by moving the abstracted information into its own class, where it can be measured separately from the concrete class underlying the execution.

3.3.5. Explicit Abstraction

The abstraction mechanisms described above all rely upon the invariant detector measuring the concrete instance when its methods are invoked, and capturing state associated with the concrete instance (directly via member variables, or indirectly via inline abstracted state or pure functions). Isolating the abstraction variables requires filtering to separate them from the concrete variables. This naturally leads to the idea of creating a separate abstraction instance that parallels the concrete instance, and to locate the abstraction variables in that *lifted* instance. If we have an interface to our invariant detector that lets us *pretend* to invoke methods on the abstraction, then we can perform invariant detection on the abstraction, obtaining results that are isolated from the concrete variables, yet informed by them.

There are other benefits to creating a separate abstraction object. One of the initial motivations for our research on abstraction involved behavioral subtype invariant detection, where we needed to create abstractions that corresponded to the superclasses of an instance, and to propagate traces to these synthetic classes. As we will detail in 3.4. Alembic), Alembic adopts this idea of creating an explicit abstraction object and applying the invariant detector to it. We will show other examples in Chapter IV that involve creating abstraction instances that do not correspond to concrete instances, and could not be solved in Daikon alone.

3.3.6. AspectJ

The most powerful and general abstraction techniques proposed above require that the concrete source code is modified in some way. This modification might be the addition of special pure functions to be used by a *purity file* or the addition of instrumentation to update inline abstracted state variables or to synthesize traces to an explicit abstraction. In any case, the requirement that source code be modified to perform analysis is problematic.

One way to address this is to use a code-weaver like AspectJ [Kiczales et al., 2001], which allows all of these code modifications to be specified in a separate set of `aspect` files. The `ajc` compiler is used instead of `javac` for compiling source files and aspect files; the compiler *weaves* the code amendments from the aspect files into the final set of compiled class files. Another benefit is that AspectJ does not require source code at all; it is able to weave code into compiled class files or libraries.

Our initial work on abstraction involved the manual creation of abstraction classes and the hand-modification of concrete methods to synthesize invocations on the abstracted methods. This was extremely tedious and inflexible, but sufficient to demonstrate the promise of abstraction. We adopted AspectJ as a way to more easily insert the required trace generation at function entry and exit, but found that its capabilities were very useful in other ways. For example, we use AspectJ to place the representation function generated by the `lift_state` clause *into* the concrete class. Effectively, we are adding an abstraction-specific pure function to the concrete class. Porting Alembic to a non-Java platform would necessitate adopting a different code-weaving facility, such as AspectC++ [Spinczyk et al., 2005].

3.4. Alembic

Alembic is a language and system for specifying and performing dynamic program analysis. Specifically, the dynamic analysis will infer behavioral specifications from executing programs by using the dynamic invariant detection mechanism upon classes that are synthesized from corresponding Alembic language declarations.

Some of the examples of abstraction presented in this dissertation could be developed manually by explicitly creating abstraction classes and explicitly inserting instrumentation into concrete classes, as we described in the preceding sections. However, the tedium and brittleness of such a solution makes it impractical. To address this, we developed Alembic to manage the generation of abstraction classes, the weaving of instrumentation, and the management of the multi-phased build and analysis process. Our experience so far has been positive; we can conceive and perform experiments rapidly by simply editing an Alembic source file and executing it.

One way to view dynamic invariant detection is as a series of processes (micro-invariant detectors) that are fed *trace streams* via some instrumentation mechanism. These trace streams contain a record of the value of arguments and state variables for all of the targeted function entry and exits. The micro-detectors read these streams and infer invariants that describe the observed patterns of behavior. Alembic generalizes this notion by providing a mechanism for *trace propagation* and *trace transformation*.

3.4.1. Using Alembic for Analysis

The first step in using Alembic for program understanding is to create an Alembic source file that describes the program to execute and the trait and view declarations to apply and analyze. For our purposes, we assume that a human creates the Alembic file, although the language is designed to be easily generatable by other tools. We also assume that the author has knowledge of the interface of the targeted class, including of any variables that might be needed for a view's lift_state clause.

The Alembic compiler is then used to parse the Alembic file, creating an execution environment and script. Executing this script results in the generation of a Java class for each trait, an AspectJ aspect for each view, and an aspect for each concrete class to be instrumented. Alembic then executes the target program in conjunction with the generated classes and aspects and Daikon's trace generation library.

The traces that result from the above execution are synthetic traces upon the abstraction class's methods, and it is these that are fed to the traditional detector (Daikon) for invariant detection. The invariants reported by this detector are then post-processed slightly for readability before being output to the user as the result of the Alembic analysis (see Figure 3.1).

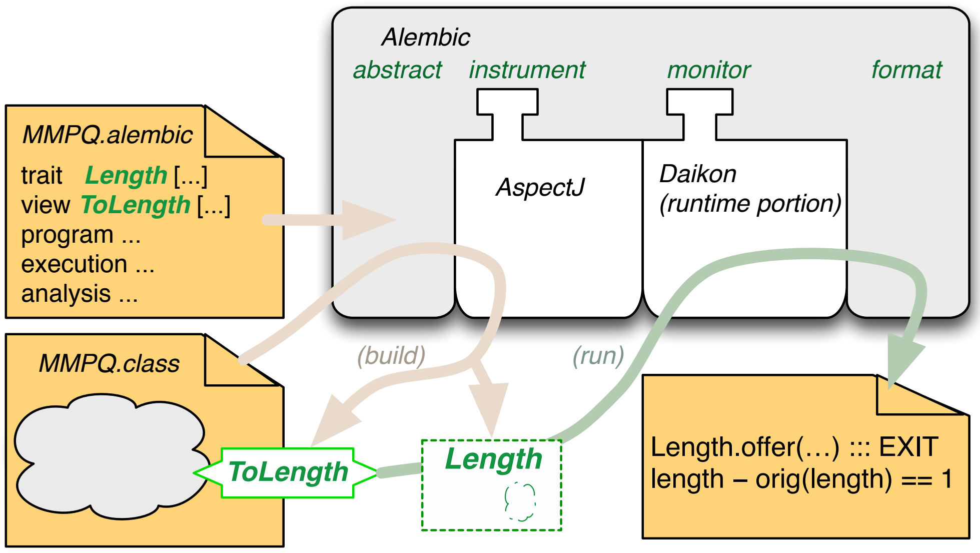


Figure 3.1. The Alembic invariant detection process. Traits defined in an Alembic file are translated into Java source files (not shown) with empty implementations. Views are inserted into the class under test through a combination of BCEL and AspectJ, and feed pseudo-events on the abstraction classes to the dynamic analysis portion of Daikon.

Abstraction via Alembic provides an additional way to analyze a module, which can often lead to improved understanding of that module. Alembic can be used to supplement or to replace traditional invariant detection. The analyst effort in constructing an Alembic file is proportional to the sophistication of the abstraction; the bookkeeping needed to apply the abstraction is hidden, providing leverage for the analyst.

In addition to reducing the amount of glue code necessary to express an abstraction, Alembic provides a syntax that is more expressive and concise than manual encoding of abstractions and the requisite aspects and adaptors. One other advantage of a DSL (domain-specific language) like Alembic is the possibility of determining errors at compilation time, based upon the global knowledge available to the alembic compiler. Alembic does not currently implement a deep error analysis prior to generating code, although this is clearly possible in the framework.

Alembic is designed to be used by an analyst by first creating an Alembic source file that specifies the target program and one or more execution configurations. Together, these will be sufficient information to build and execute the target program. We assume that the author has some knowledge of the interface of the targeted class, including of any variables or accessor methods that might be needed for the view's lift_state function.

We begin by declaring one or more traits in an Alembic file (see Listing 3.1). These will result in the generation of corresponding abstraction classes. If the use clause is specified, then a *static abstraction* is performed on the concrete class. This build-time operation populates the generated abstraction class with methods that are based upon the concrete class's public methods.

Finally, one or more views are declared in the Alembic file. These specify the abstraction functions via the lift_state clause, as well as when and how they should be applied to the concrete invocations.

The alembic compiler is then used to parse the Alembic file, creating a runtime script and environment (a build directory and a makefile). Executing these results in the generation of a Java class for each trait and an AspectJ aspect for each view. After all necessary code is generated, the system executes the target program in conjunction with the generated classes and aspects and Daikon's trace generation library.

The traces that result from the above execution are synthetic traces upon the abstraction class's methods, and it is these that are fed to the traditional detector (Daikon) for invariant detection. The invariants reported by this detector are then post-processed slightly for readability before being output to the user as the result of the Alembic analysis.

We can conceive of Alembic applications where the .alembic file is generated by another tool, rather than hand-crafted by an analyst. For example, it may be possible to derive view and trait information from a JML specification so that invariant detection can be performed on the JML model fields. Or perhaps a preprocessor could generate .alembic files and .patch files to automatically instrument loops and perform loop analysis upon them (see 4.3.2. Abstract Functions and Loop Abstraction).

3.4.2. Structure of the Alembic Language

The Alembic syntax is primarily declarative, although there may be embedded Java code to implement representation functions and other customizable aspects of Alembic. An Alembic file consists of a series of declarations of the following constructs:

trait This declares state variables and method definitions and results in the generation of a Java class that represents an abstraction and serves as the target of dynamic invariant detection. The use clause enables the copying of method signatures from a concrete class, simplifying the abstraction of concrete classes as traits.

view This represents the *lift* operation that translates executions on a concrete object into executions on an abstract trait. Minimally, a view must specify a lift clause indicating the underlying concrete class to instrument and the target trait to use

Listing 3.1. An Alembic file that abstracts invocations of MMPQ into two traits, Length and Sorted (1 of 2).

```
trait Length
[
    use com.google.common.collect.MinMaxPriorityQueue;
    state int length;
]

trait Sorted
[
    use com.google.common.collect.MinMaxPriorityQueue;
    state int[] sorted;
]

view ToLength
[
    lift com.google.common.collect.MinMaxPriorityQueue<?> to SortedObjects;
    trait Length;
    lift_state
    {
        abs.length = this.size();
    }
]

view ToSorted
[
    lift com.google.common.collect.MinMaxPriorityQueue<?> to SortedObjects;
    lift_state
    {
        java.util.Arrays.sort( abs.sorted = this.toArray() );
    }
]
```

for synthetic traces. The view also provides a lift_state clause to perform state abstraction, a lift_method clause to perform method transformation.

program This declaration identifies a program or library to be built, via the builddir option, and specifies the traits and views to include via the traits and views clauses.

execution An execution declaration specifies a particular, parametrized running of a previously declared program referred to via the program clause. The rundir and main clause indicate which main program should be used and where it should be executed. The args clause allows runtime arguments to be specified in the Alembic file that will be passed to the program execution.

analysis The analysis keyword is currently necessary in Alembic, but has no useful options other than referring to a prior execution via the execution keyword.

Alembic distinguishes between the static interface defined by trait and the dynamic transformation defined by view, rather than having a single construct that manages both the representation and transformation. This is to support abstraction patterns where multiple concrete methods or classes can be lifted to a single trait or a single method within a trait. Together, these separate constructs allow a rich variety of abstractions to be constructed, upon which an invariant detector can perform its inference.

An Alembic script includes blocks defining and applying explicit abstractions, and directives controlling the execution and analysis processes. A simple form of block inheritance allows attributes declared in one block to be used in another, simplifying construction of Alembic scripts with multiple abstractions and views targeting a common class.

$$\begin{aligned}
\langle \textit{Script} \rangle &::= (\langle \textit{Abstraction} \rangle \mid \langle \textit{Control} \rangle)^* \\
\langle \textit{Abstraction} \rangle &::= \langle \textit{Trait} \rangle \mid \langle \textit{View} \rangle \\
\langle \textit{Control} \rangle &::= \langle \textit{Program} \rangle \mid \langle \textit{Execution} \rangle \mid \langle \textit{Analysis} \rangle
\end{aligned}$$

The *shape* of an abstraction is defined separately from the *lifting* mechanism that extracts it from concrete state; in some cases, we wish to lift multiple sources to a common abstraction so that the same abstract state, such as a set of ordered pairs, can represent different concrete structures, such as a hash table and search tree implementations of a dictionary. The trait declaration can contain state variables and a subset of methods derived from a concrete Java class.

$\langle$*Trait*$\rangle$ ::= `trait` *id* [: *id*]
[($\langle$*TraitAttr*$\rangle$;)*]

$\langle$*TraitAttr*$\rangle$ ::= `state` *type-id id*
| `use` *classname* : *regexp*
| `trait` *id*
| `class` *classname*
| `filter` *regexp*

The class and filter attributes of a trait create methods in the abstraction class mirroring the public methods in a concrete class; variants specialized to extracting method definitions from jar files and source code libraries are also provided. filter allows a subset of methods matching a regular expression to be copied. The syntax use <class> : <filter> is a convenient way to specify these options.

A view defines the mapping from the concrete class (the class and filter attributes) to a trait; it is effectively a functor from the concrete class to the abstract class generated from the trait. The lifting function from concrete to abstract state is specified operationally as Java code in the lift_state attribute. The optional lift_method clause enables explicit, dynamic control of the abstraction process, which allows values to be transferred between state, argument and result variables prior to invoking a method on the trait.

$\langle$*View*$\rangle$ ::= `view` *id* [: *id*]
[($\langle$*ViewAttr*$\rangle$;)*]

$\langle$*ViewAttr*$\rangle$ ::= `trait` *id*
| `as_method` *method-name*
| `class` *class-name*
| `lift_state` { *Java code* }
| `lift_method` *method-name*
[$\langle$*EnterGlue*$\rangle$ $\langle$*ExitGlue*$\rangle$]

The control part of an alembic script manages the execution and analysis workflow. A program specifies the subset of declared traits or views to be applied, as well as build and environment options for the concrete program; an execution specifies which program to run, along with its parameters; and an analysis determines what kind of invariant detection is performed for each execution.

```
⟨Program⟩        ::= program id [ : id ]
                     [ (⟨ProgramAttr⟩ ; )* ]

⟨ProgramAttr⟩    ::= builddir directory path
                     | traits string  | views string

⟨Execution⟩      ::= execution id [ : id ]
                     [ (⟨ExecutionAttr⟩ ; )* ]

⟨ExecutionAttr⟩  ::= rundir directory-path
                     | program id
                     | main class-name
                     | arguments string

⟨Analysis⟩       ::= analysis id [ : id ]
                     [ (⟨AnalysisAttr⟩ ; )* ]

⟨AnalysisAttr⟩   ::= classes string
                     | execution id
```

3.4.3. Performance Characteristics

Our experience with Alembic so far has shown that it introduces no significant overhead to the use of dynamic invariant detection. In this section, we break down the various factors that could potentially contribute to Alembic's overhead. There are three primary components that impact the user's experience using Alembic:

Building This is the time required to compile the Alembic file, generate Java classes and AspectJ aspects, and to compile these components and the main program specified in the program clause of the Alembic file. This is a relatively fixed cost, proportional to the number of signatures copied or instrumented via use and lift clauses. We do not consider it a significant factor affecting the tool's utility.

Execution This is the time required to run the target program, plus the overhead of instrumentation, abstraction, and trace generation. Alembic and Daikon both must bear the cost of instrumentation and trace generation; Alembic's representation functions (lift_state) add an additional cost that Daikon does not share. However, this extra cost may be offset by a reduced amount of data that must be emitted as traces; several of our Guava examples in Chapter IV demonstrate this reduction in trace size and overall execution time. In principle, one might write a very complex lift_state function that is very expensive; however, our experience has been that in practice the execution time of the representation function is insignificant compared to the impact of trace size on execution time.

Analysis This is the time required for the reading and interpretation of generated traces. Alembic plays no part in this phase, except in its influence on the size and content of the traces. As with the Building component, we treat this as a fixed cost and do not measure it further.

We focus our performance analysis on the Execution component above, because it is the only place where non-linear Alembic overhead could jeopardize its utility. We first compare the execution time for the MMPQ unit test program under various levels of invariant detection and abstraction, which we show in Table 3.1. We chose this test program because it is the most robust of the Guava tests and has a relatively long execution time. The timing code is inserted into the main program after any initialization has been performed.

Table 3.1. Execution overhead for different levels of invariant detection on the MMPQ unit test. The time reported is elapsed wall clock time in seconds.

Level	**Description**	**Time**
Raw Test	Unmodified execution of the program without Daikon or Alembic.	0.5
Daikon Test	Daikon instrumentation and tracing, filtered to include only public, non-static methods. No Alembic used.	29.2
Stateless	Abstraction of MMPQ to a trait with no state or lift_state used.	9.4
Integer	Abstraction of MMPQ to a trait with single integer state variable with trivial lift_state used.	11.8
Pair	Abstraction of MMPQ to a trait with two integer state variables with trivial lift_state used.	13.1
Length	Abstraction of MMPQ to an integer length.	11.8
SortedObjects	Abstraction of MMPQ to a sorted array.	20.8

The times in Table 3.1 reveal that the default Daikon instrumentation of MMPQ's public, non-static methods adds almost 29 seconds to the raw unit test cost of 0.5 seconds. When we consider the various Alembic abstractions, we see that the primary factor in execution time is the size of trace generated, which is based upon the number and types of the variables traced. A scalar state variable will have a small trace footprint, whereas an array of arrays will have a potentially much larger trace.

If we look closer at the phases of processing and abstraction that Alembic uses, we can isolate the various costs involved, which we report in Table 3.2. The elapsed time reported for each was gathered by incrementally modifying the Alembic code generator to add successively more layers of processing, until the ultimate generation of abstract traces. From the table, it is clear that the bulk of the overhead is in the trace generation. The costs of the AspectJ instrumentation and the lift_state function are overwhelmed by the cost of trace generation. This fact is in Alembic's favor, because Alembic provides a way to control the size of the trace file in more selective ways that Daikon alone.

Table 3.2. Execution overhead for phases of Alembic on the MMPQ unit test (SortedObjects trait), broken down into the steps involved in instrumentation, abstraction, and trace synthesis. The time reported is elapsed wall clock time in seconds.

Phase	**Description**	**Time**
Raw Test	Unmodified execution of the program without Daikon or Alembic.	0.5
No Trace Daikon	Executed under control of Daikon, but no traces emitted.	1.0
AspectJ	Aspect simply proceeds and perform no further abstraction or tracing.	1.6
Adaptor	Aspect manages the dispatch and interlocks that enable multiple abstractions, but no lifting or trace generation occurs.	1.6
Lift	lift_state is invoked, but no traces are generated.	2.6
Trace	Traces are generated from the abstracted invocations.	20.8

3.4.4. Modifications to Daikon

Alembic relies upon the ability to fool the back-end invariant detector into thinking that a method was invoked on an abstraction object, so that invariant detection is performed upon these synthetic invocations. In the case of Daikon, we used the methods Runtime.enter and Runtime.exit within the daikon.chicory package. Daikon's instrumenter, Chicory, will ordinarily insert Java bytecode to call Runtime.enter at the beginning of an instrumented method body, and to call Runtime.exit before each return in the method body. Alembic's generated abstraction classes contain a pair of methods liftEntry_someMethod and liftExit_someMethod for every method someMethod that exists in the trait. These *lift* methods are responsible for converting argument types into the wrappers necessary for Daikon, and then calling the Daikon enter/exit methods to generate traces.

One requirement for Alembic to use Daikon's Runtime.enter and Runtime.exit is that we provide a *method index*, which is an integer index identifying a unique class and method combination. These method indexes are generated within Daikon (specifically, Chicory) when it loads class definitions to be analyzed for instrumentation. We made a small modification to Daikon such that it maintains a hash table that can efficiently turn a class and method name into a method index and we use this when initializing Alembic's generated abstraction classes to compute the method indexes necessary to synthesize invocations.

One additional improvement we made to Daikon's Chicory tool was to enable filtering of the getClass() pseudo-variable via the Chicory –omit-var command-line option. This variable was not being correctly excluded from traces, so we made a modification to Chicory so that it honored the –omit-var=getClass we use for Alembic and Daikon.

3.5. Non-Daikon and Non-Java Implementation

Although we are using Java and AspectJ for the current implementation of Alembic, the technique is amenable to any language where we can obtain the value of visible variables at function entry and exit. Instrumentation for other languages would require the use or development of something like AspectJ. There are existing aspect-oriented programming (AOP) systems for many languages, including C++ [Spinczyk et al., 2005], that could provide the needed mechanism. Even in the absence of AOP support, all that Alembic really needs is a way to insert code into methods at entry and exit points; this is achievable with existing tools, although AOP makes the solution easier to support.

Listing 3.2. An Alembic file that abstracts invocations of MMPQ into two traits, Length and Sorted (2 of 2).

```
program MMPQP
[
    builddir "../../examples/MMPQ/";
    views "ToLength ToSorted";
]

execution MMPQPE
[
    program MMPQP;
    rundir "../../examples/MMPQ/";
    main "MMPQUnitTest";
]

analysis MMPQPEX1
[
    execution MMPQPE;
]
```

CHAPTER IV

EXAMPLES, EVALUATIONS AND RESULTS

Portions of the material in this chapter were co-developed with Yannis Smaragdakis and Michal Young for an unpublished paper. They helped with the experimental design and data presentation.

In the previous chapter, we presented Alembic and illustrated some of its capabilities with simple examples. We continue in this chapter with a more systematic evaluation of Alembic as well as a presentation of some of the more interesting abstraction capabilities possible.

Our research is oriented towards the exploration of abstraction techniques, some of them immediately practical, and some of them we considered because they seemed to follow naturally from our work or seemed to be promising paths for future researchers. This chapter begins with a description of the methodology, software and environments used for the experiments presented in the remaining sections. We use the term *experiment* loosely here; it is intended to indicate a usage scenario consisting of a goal (e.g., module understanding or clarification), execution and subsequent analysis.

The first type of scenario we consider is oriented around simple state abstraction of container classes, including several representative classes from the Guava Java Library [Google, 2011]. We will show how we can *lift* potentially opaque implementation state into simple and transparent data structures that are more suitable for invariant detection.

We then look at some of the more exotic abstraction techniques enabled by Alembic, including loop invariants and history constraints. These examples are still preliminary, but illustrate the potential for Alembic in enabling invariant detection to be applied in a wider context. We conclude this chapter with a discussion of Alembic's performance implications by dissecting the overhead of the various phases of instrumentation and abstraction.

4.1. Evaluation Methodology

Our research introduces an enhanced form of dynamic invariant detection, where the user can mine invariants associated with abstracted versions of targeted concrete classes. In the following sections of this chapter, we will present a series of usage scenarios for this type of enhanced detection. We make the case that this new analytic technique, and its implementation in Alembic, can provide better insight into a module's behavior than can a detector observing only concrete behavior. This better insight may sometimes be in the form of a reduced set of invariants for the human analyst to study, or in the form of invariants that are expressed in terms of methods and variables suited to the analyst's desired level of abstraction and understanding. In some cases, abstraction is used simply to transform an opaque data structure into something more visible and manipulable by a dynamic invariant detector.

These characteristics are inherently subjective, so our case for the value of abstraction-enhanced invariant detection will be primarily based upon showing examples of Alembic in action. We will, however, adopt a few metrics to attempt to quantify this notion of *better invariant detection*.

Alembic Script Lines The size of the input Alembic file in terms of source lines, ignoring blank lines and lines with only brackets. This usually reflects the effort that an analyst needs to use to extract useful information from the invariant detection tool. We treat the Daikon script size as a constant to favor Daikon in our comparison; however, we actually must add –ppt-omit-pattern clauses to the Daikon script in order to filter out private and static methods so that the trace file size and execution time comparisons are meaningful.

Source Lines The number of source lines of the class under analysis. This is a crude surrogate for the class's complexity, but it is easily obtained and is a widely accepted metric.

Methods Instrumented The number of methods which were instrumented, even if no invariants were reported. In our invariant listings, we ignore methods for which no invariants are reported, even if we requested their instrumentation. Usually, an empty report indicates that the methods were never called, or that no detectable pattern was observed (i.e., an invariant of true) in the invocations.

Invariants Reported The number of invariants reported for public, non-static methods and the object invariant. In the case of Daikon, we use explicit –ppt-select-pattern qualifiers to include only these methods to ensure fairness in comparison with Alembic.

Number of State Variables The number of visible variables available for use in invariant expressions. Generally, the number of invariants scales combinatorially with the number of visible variables.

Trace File Size For a given test program of an instrumented module, the size of the trace file generated is indicative of the amount of data being captured, and of the amount of data that must be analyzed by the detector. We indicate this file size in units of megabytes. Daikon uses compressed files for transport and archive, but generates and consumes uncompressed files; we show both sizes in most cases below.

Execution Time Each test driver we use captures the wall-clock execution time via Java's System.nanoTime() function and reports this time in seconds. For most tests, we encapsulate the central driver loop in timing code, and leave any one-time initialization code outside of the timing block.

Ideally, the Alembic input and output quantities above would be smaller than the corresponding Daikon quantities for each problem we present. However, there are cases where the additional insight made available by Alembic is impossible with Daikon alone, or where an increase in the effort crafting an Alembic file pays off in a more focused invariant report.

4.1.1. Hardware and Software

The results in this section were obtained using the following hardware and software environment:

- Apple MacBook Pro (2009 aluminum unibody model)
- Intel Core 2 Duo 2.66GhZ Processor
- 4GB Memory
- Java SE 1.6
- Mac OS X 10.7.2 (Lion)
- Daikon version 4.6.4
- AspectJ Compiler 1.6.12
- ANTLR v3.3
- Python 2.7.1
- Guava Release 09[1]

4.1.2. Experimental Setup

Only public instance methods will be instrumented for both Alembic and Daikon experiments. Daikon instruments all methods by default and there is no convenient way to instruct it to operate on public methods only, so we simply exclude the non-public or static methods from the metrics we report. Alembic traits, on the other hand, are generated such that only the intended methods are instrumented.

[1]Our original Guava work began with Guava release 09, which we have continued to use. The current version of Guava as of January 2012 is Guava Release 11.

4.2. Guava Collection Classes

One of the common usage scenarios we anticipate with Alembic is where a stateful concrete object is to be analyzed by creating an abstracted version of the object that has abstracted state variables, and method signatures that match the public, non-static methods of the concrete class[2]. We call this type of abstraction *state abstraction*, and Alembic is well-suited to expressing experiments that perform invariant detection on abstractions of stateful objects. The subsections below describe individual use cases where we choose a container class from the Guava library and develop a few abstractions to apply. Our intent isn't to show that the particular abstractions we choose are the best or smallest, but to show that the analyst has some control over the invariant detection process, and that *playing around* with different abstractions is quite easy in Alembic.

We begin below with a detailed continuation of the MMPQ example from Chapter I, followed by analyses of representative container classes from the Guava library. The Guava library [Google, 2011] is a set of Java classes developed by Google for its Java-based applications. We chose this library because it is a real production library actively used in widely deployed applications, because it has easily available source code, and because it provides a set of container classes that are highly optimized, but mostly inscrutable via traditional invariant detection. In addition to the container classes (com.google.common.collect.*), Guava provides a rich set of classes that make Java programming more powerful and pleasant. These include:

- Precondition testing mechanisms
- Better handling and of exceptions and errors
- Cache management utilities
- Concurrency abstractions to simplify writing correct concurrent code
- String manipulation utilities
- Math utilities

We focus on the collection classes for our simple state abstraction examples because collections have optimized and usually inscrutable data structures as their implementation of state, making a traditional concrete invariant detector less useful for their analysis. However, these classes are ideal candidates for applying abstraction techniques which lift this optimized concrete state into simpler data structures more accessible to a dynamic

[2]In the next section, we will consider more exotic abstraction patterns where the abstraction may have a distinct set of methods from the concrete class, and where multiple concrete methods are lifted to a common abstract method.

invariant detector. Later examples in this chapter will use additional abstraction techniques for other, non-collection, classes.

4.2.1. MinMaxPriorityQueue

The MinMaxPriorityQueue (MMPQ) example from Chapter I was a simple one, where we created an abstraction with a single offer method whose signature matched the concrete MMPQ.offer method (Listing 1.5). We used a lift_state clause to declare our representation or abstraction function, which abstracted the concrete queue as a sorted array of objects. The resulting invariants revealed the behavior of offer in terms of the trait SortedObjects. Although we only had our Alembic file from Chapter I consider the offer method, this was simply an optimization to reduce the amount of trace data generated.

In this section, we will repeat the experiment without the offer filter, so that all of the public methods in MMPQ will be abstracted into the SortedObjects trait. In addition, we will replace our handcrafted test driver, MMPQTest, with a new test driver, MMPQUnitTest, that runs most of the unit tests from the Guava distribution[3]. We do this to reduce any perceived bias in our testing regime, as well as to provide us with richer tests for free. The remainder of the Guava tests in this chapter will be instrumenting all public methods and using the Guava-provided unit tests when possible.

We summarize the important metrics associated with this comparison experiment in Table 4.1, where we present the common attributes of the test as well as the metrics used to compare Daikon and Alembic. Note the reduced amount of invariants for the Alembic run, as well as the reduced execution time. These quantities will not always be lower for Alembic, depending on the size and complexity of the abstraction; however, they do show that reasonable results can be obtained without significant overhead difference from Daikon alone. The Guava experiments we present in this chapter all of Alembic execution times that are the same order of magnitude as the corresponding Daikon execution times, and sometimes smaller.

The Alembic file in Listing 4.1 will instrument all of the public methods in MMPQ and will exercise the module using the more thorough unit test in Listing 4.2. The resulting invariants are displayed in Listing 4.3 and Listing 4.4 below. Note that Daikon uses the suffix :::ENTER to represent an entry program point, and :::EXIT for an exit point, which syntax Alembic passes through to its final report.

[3]The unit tests consist of a main test function that invokes a series of roughly 35 Guava-provided tests of different MMPQ features and capabilities. The amount of time and space used by Daikon for some of these tests was prohibitive, so we opted to not use them for our comparison, although Alembic would likely perform better in these cases. The remainder of the Guava tests use unmodified unit tests from Guava.

Listing 4.1. The MMPQ.alembic file, containing definitions required to lift method invocations on MMPQ to method invocations on the SortedObjects trait, via the ToSortedObjects view.

```
trait SortedObjects
[
    use com.google.common.collect.MinMaxPriorityQueue;
    state Object[] sorted;
]

view ToSortedObjects
[
    lift com.google.common.collect.MinMaxPriorityQueue<?> to SortedObjects;
    lift_state
    {
        java.util.Arrays.sort( abs.sorted = this.toArray() );
    }
]

program MMPQP
[
    builddir "../../examples/MMPQ/";
    views "ToSortedObjects";
]

execution MMPQPE
[
    program MMPQP;
    rundir "../../examples/MMPQ/";
    main "MMPQTest";
]

analysis MMPQPEX1
[
    execution MMPQPE;
]
```

Listing 4.2. A fragment of our test driver MMPQUnitTest which invokes most of the battery of Guava-provided unit tests to exercise MMPQ. Some Guava-provided tests were prohibitively expensive for Daikon to analyze, and were therefore omitted from our test driver for both Alembic and Daikon.

```
import com.google.common.collect.*;
// Guava−provided unit tests invoked below
testCreation_simple();
testCreation_comparator();
testCreation_expectedSize();
testCreation_expectedSize_comparator();
testCreation_maximumSize();
testCreation_comparator_maximumSize();
testCreation_expectedSize_maximumSize();
testCreation_withContents();
testCreation_comparator_withContents();
testCreation_expectedSize_withContents();
testCreation_maximumSize_withContents();
testCreation_allOptions();
testSmall();
testSmallMinHeap();
testRemove();
testContains();
testIteratorPastEndException();
testIteratorConcurrentModification();
testIteratorRegressionChildlessUncle();
testInvalidatingRemove2();
testInvalidatingRemove2();
testIteratorInvalidatingIteratorRemove();
testIteratorInvalidatingIteratorRemove2();
testCreateWithCapacityAndOrdering();
try { testIteratorTester(); } catch ( Exception e ) {}
try { testIteratorTesterLarger(); } catch ( Exception e ) {}
testRemoveAt();
testCorrectOrdering_regression();
testCorrectOrdering_smallHeapsPollFirst();
testCorrectOrdering_smallHeapsPollLast();
testCorrectOrdering_73ElementBug();
testRegression_dataCorruption();
testIsEvenLevel();
try { testNullPointers(); } catch ( Exception e ) {}
```

Listing 4.3. Alembic invariants for MMPQ as exercised by MMPQUnitTest using the view ToSortedObjects to abstract the concrete object to the SortedObjects trait. Program points without reported invariants have been excluded from this listing (1 of 2).

```
SortedObjects:::OBJECT
  this has only one value
  this.sorted != null
  this.sorted[] elements != null

SortedObjects.add(java.lang.Object):::ENTER
  element != null

SortedObjects.add(java.lang.Object):::EXIT
  size(this.sorted[])−1 == orig(size(this.sorted[]))
  return == true
  size(this.sorted[]) >= 1
  orig(element) in this.sorted[]

SortedObjects.addAll(java.util.Collection):::ENTER
  this.sorted[] == []

SortedObjects.addAll(java.util.Collection):::EXIT
  return == true
  size(this.sorted[])−1 > orig(size(this.sorted[]))

SortedObjects.comparator():::ENTER
  size(this.sorted[]) one of { 0, 6 }

SortedObjects.comparator():::EXIT
  this.sorted[] == orig(this.sorted[])
  size(this.sorted[]) one of { 0, 6 }

SortedObjects.iterator():::EXIT
  this.sorted[] == orig(this.sorted[])

SortedObjects.offer(java.lang.Object):::ENTER
  element != null

SortedObjects.offer(java.lang.Object):::EXIT
  size(this.sorted[])−1 == orig(size(this.sorted[]))
  return == true
  size(this.sorted[]) >= 1
  orig(element) in this.sorted[]
```

Listing 4.4. Alembic invariants for MMPQ as exercised by MMPQUnitTest using the view ToSortedObjects to abstract the concrete object to the SortedObjects trait. Program points without reported invariants have been excluded from this listing (2 of 2).

```
SortedObjects.peek():::EXIT
  this.sorted[] == orig(this.sorted[])

SortedObjects.peekLast():::EXIT
  this.sorted[] == orig(this.sorted[])

SortedObjects.poll():::EXIT
  size(this.sorted[]) <= orig(size(this.sorted[]))
  size(this.sorted[]) >= orig(size(this.sorted[]))−1
  size(this.sorted[])−1 <= orig(size(this.sorted[]))−1

SortedObjects.pollFirst():::ENTER
  size(this.sorted[]) >= 1

SortedObjects.pollFirst():::EXIT
  size(this.sorted[]) == orig(size(this.sorted[]))−1
  return in orig(this.sorted[])

SortedObjects.pollLast():::EXIT
  size(this.sorted[]) <= orig(size(this.sorted[]))
  size(this.sorted[]) >= orig(size(this.sorted[]))−1
  size(this.sorted[])−1 <= orig(size(this.sorted[]))−1

 SortedObjects.size():::EXIT
  this.sorted[] == orig(this.sorted[])
  return == size(this.sorted[])
  return == orig(size(this.sorted[]))
```

Table 4.1. Alembic and Daikon analyses of MMPQ.

Abstraction	MinMaxPriorityQueue (MMPQ) as a sorted array of objects, in ascending order.
Class Size (lines)	947
Public Methods	16
Raw Execution (seconds)	< 1.0
Alembic Script (lines)	18

	Daikon	**Alembic**
Number of Invariants	352	34
Trace file compressed (M)	4.5	6.5
Trace file uncompressed (M)	248	153
Execution Time (sec)	30	22

We can use Daikon to observe the same unit test execution by using the commands in Appendix A (Listing A.1). This generates over 350 invariants, most of which are not useful for specification purposes or much of anything. The invariants are referring to internal variables that are used in the implementation of a class, and not to variables that clearly express the desired *abstract* class. The state variables needed to implement a class efficiently is often not the same set as the state variables that best express the abstract behavior of the class, which is one reason why Alembic can be useful in presenting an abstracted class for invariant detection.

We show the first two pages of the ten pages of Daikon output for the same test in Listing 4.5 through Listing 4.6 below. As can be seen in the Daikon output for this example, most of the invariants are useless for us to infer the behavior of this class, or even whether it is behaving according to its documented specification. On the other hand, the Alembic invariants nicely express the behavior of the methods in terms of our desired abstraction as a sorted list of objects. The entirety of the Daikon output has been placed in several listings in Appendix B (Listing B.1). For subsequent Guava examples, we will not display the Daikon output in its entirety for space reasons.

4.2.2. HashBasedTable

The next experiment we describe involves the Guava collection class HashBasedTable (HBT). We summarize our results in Table 4.2, and present details about the comparison

Listing 4.5. Partial listing of Daikon invariants for methods of MMPQ exercised by MMPQTest (1 of 2).

```
MMPQ:::OBJECT
  this.minHeap != null
  this.maxHeap != null
  this.maximumSize one of { 42, 2147483647 }
  this.queue != null
  this.size >= 0
  this.modCount >= 0
  this.maximumSize > this.size
  this.maximumSize > this.modCount
  this.maximumSize != MMPQ.EVEN_POWERS_OF_TWO
  this.maximumSize > MMPQ.ODD_POWERS_OF_TWO
  this.maximumSize > MMPQ.DEFAULT_CAPACITY
  this.maximumSize > size(this.queue[])
  this.size <= this.modCount
  this.size < MMPQ.EVEN_POWERS_OF_TWO
  this.size > MMPQ.ODD_POWERS_OF_TWO
  this.size <= size(this.queue[])
  this.modCount < MMPQ.EVEN_POWERS_OF_TWO
  this.modCount > MMPQ.ODD_POWERS_OF_TWO
  MMPQ.EVEN_POWERS_OF_TWO > size(this.queue[])
  MMPQ.ODD_POWERS_OF_TWO < size(this.queue[])−1

MMPQ.add(java.lang.Object):::ENTER
  this.maximumSize == 2147483647
  element != null

MMPQ.add(java.lang.Object):::EXIT
  this.minHeap == orig(this.minHeap)
  this.maxHeap == orig(this.maxHeap)
  this.maximumSize == orig(this.maximumSize)
  this.queue[this.size−1] == this.queue[orig(this.size)]
  this.maximumSize == 2147483647
  this.size >= 1
  this.modCount >= 1
  return == true
  this.queue[this.size−1] != null
  this.maximumSize > orig(this.size)
  this.maximumSize > orig(this.modCount)
  this.maximumSize > orig(size(this.queue[]))
  orig(element) in this.queue[]
  this.size − orig(this.size) − 1 == 0
```

Listing 4.6. Partial listing of Daikon invariants for methods of MMPQ exercised by MMPQTest (2 of 2).

```
  this.size != orig(this.modCount)
  this.modCount > orig(this.size)
  this.modCount − orig(this.modCount) − 1 == 0
  MMPQ.EVEN_POWERS_OF_TWO > orig(this.size)
  MMPQ.EVEN_POWERS_OF_TWO > orig(this.modCount)
  MMPQ.ODD_POWERS_OF_TWO < orig(this.size)
  MMPQ.ODD_POWERS_OF_TWO < orig(this.modCount)
  orig(this.size) <= size(this.queue[])−1
  orig(this.modCount) != size(this.queue[])
  size(this.queue[]) >= orig(size(this.queue[]))
  size(this.queue[])−1 != orig(size(this.queue[]))
  size(this.queue[])−1 >= orig(size(this.queue[]))−1

MMPQ.addAll(java.util.Collection):::ENTER
  this.size == this.modCount
  MMPQ.DEFAULT_CAPACITY == size(this.queue[])
  this.queue[this.size] == this.queue[MMPQ.DEFAULT_CAPACITY−1]
  this.maximumSize == 2147483647
  this.queue[] contains only nulls and has only one value, of length 11
  this.queue[] elements == null
  this.size == 0
  this.queue[] elements == this.queue[this.size]

MMPQ.addAll(java.util.Collection):::EXIT
  this.minHeap == orig(this.minHeap)
  this.maxHeap == orig(this.maxHeap)
  this.maximumSize == orig(this.maximumSize)
  this.size == this.modCount
  this.queue[this.size] == orig(this.queue[post(MMPQ.DEFAULT_CAPACITY)−1])
  this.queue[this.size] == orig(this.queue[this.size])
  this.queue[this.size] == orig(this.queue[this.modCount])
  this.maximumSize == 2147483647
  return == true
  size(this.queue[]) one of { 11, 24, 50 }
  this.queue[this.size] == null
  this.size != MMPQ.DEFAULT_CAPACITY
  this.size > orig(this.size)
  this.size < size(this.queue[])−1
  this.size != orig(size(this.queue[]))−1
  MMPQ.DEFAULT_CAPACITY <= size(this.queue[])
  MMPQ.DEFAULT_CAPACITY != size(this.queue[])−1
```

below. The HBT class represents a two-dimensional table whose underlying storage is a HashMap. The Guava version of HBT has the bulk of its implementation in a hidden (package-private) superclass, StandardTable; the public class HBT is primarily a series of wrapper methods that invoke the corresponding superclass method. Both Alembic and Daikon are able to see into that superclass via bridge methods, which is why Daikon reports the invariants on StandardTable, rather than HashBasedTable, as we will see in the invariant listing below. We include StandardTable's source code when we count source lines in the summary in Table 4.2.

Table 4.2. Alembic and Daikon analyses of HBT.

Abstraction	HashBasedTable (HBT) as parallel arrays of rows, columns, and values (RCV).	
Class Size (lines)	1350 (with StandardTable)	
Public Methods	22	
Raw Execution (seconds)	< 1.0	
Alembic Script (lines)	34	
	Daikon	**Alembic**
Number of Invariants	434	253
Trace file compressed (M)	0.241	0.107
Trace file uncompressed (M)	1.4	1.2
Execution Time (sec)	2	3

When we apply Daikon to the put method of HBT, it reveals nothing of interest to us. This is because the only instance variables are two internal HashMaps that are inscrutable to Daikon; Daikon is unable to inspect the embedded HashMap instances within HBT. The output from Daikon for the put method can be seen in Listing 4.7.

Although the set of invariants reported by Daikon is relatively small, they reveal no information about the behavior of the HashBasedTable. The bulk of the state is hidden in data structures that are deeper than Daikon's default. Even if we increased the Daikon search depth with the –nesting-depth option, it would likely generate a larger batch of invariants that were equally difficult to read and understand.

The above problem of limited depth and opaque data structures is one of the reasons we created Alembic. We can use a simple Alembic abstraction to capture the table's cells into three parallel arrays corresponding to the *rows*, *columns* and *values* of the table; these arrays are more suited to Daikon's inference, as well as human understanding. We call the trait RCV (row, column, value), and place the trait and its view in an HBT.alembic file as in Listing 4.8 below; we don't show the program, execution and analysis because they are boilerplate text unrelated to abstraction.

We do not include the full output from the Daikon execution; as we showed above with the put method, it is mostly useless data. In Listing 4.9, we show the put and remove output from the Alembic run, as it is representative of the type of information available via our RCV trait.

4.2.3. TreeBasedTable

The Guava TreeBasedTable class represents a two-dimensional table whose rows and columns are ordered. We use the same RCV abstraction on the TreeBasedTable (TBT) class that we used on the previous HBT class. Not surprisingly, the results are similar. It is most likely that any differences are due to differences in the Guava unit tests. We summarize the comparison between Daikon and Alembic for this experiment Table 4.3, and present details about the comparison below.

Similar to HBT, the Guava version of TBT is unusual in that the bulk of the implementation lies in a hidden (package-private) superclass, StandardRowSortedTable, which itself extends the hidden StandardTable; the public class TBT is primarily a series of wrapper methods that invoke the corresponding superclass method. Both Alembic and Daikon are able to see into that superclass via bridge methods, which is why Daikon reports the invariants on StandardTable, rather than TreeBasedTable, as we will see in the invariant listing below. We include StandardTable's and StandardRowBasedTable's source code when we count source lines in the summary in Table 4.3.

When we apply Daikon to the put method of TBT, it reveals nothing of interest to us. This is because the only instance variables are two internal HashMaps that are inscrutable by Daikon; Daikon is unable to inspect the embedded HashMap instances within TBT. The output from Daikon for the put method can be seen in Listing 4.10; it is identical to that for HBT.

Although the set of invariants is relatively small, they reveal no information about the behavior of the TreeBasedTable. The bulk of the state is hidden in data structures that are deeper than Daikon's default. As with HBT above, increasing the Daikon search depth with the –nesting-depth option will not help; it will likely generate a larger batch of invariants that are equally difficult to read and understand.

Listing 4.7. Daikon invariants for the put method of HBT exercised by HBTUnitTest.

```
StandardTable.put(java.lang.Object, java.lang.Object, java.lang.Object):::ENTER
 this.cellSet == null
 this.columnKeySet == null
 this.rowMap == null
 rowKey != null
 columnKey != null
 value != null

StandardTable.put(java.lang.Object, java.lang.Object, java.lang.Object):::EXIT
 this.backingMap == orig(this.backingMap)
 this.factory == orig(this.factory)
 this.cellSet == orig(this.cellSet)
 this.rowKeySet == orig(this.rowKeySet)
 this.columnKeySet == orig(this.columnKeySet)
 this.values == orig(this.values)
 this.rowMap == orig(this.rowMap)
 this.columnMap == orig(this.columnMap)
 this.cellSet == null
 this.columnKeySet == null
 this.rowMap == null
```

Table 4.3. Alembic and Daikon analyses of TBT.

Abstraction	TreeBasedTable (TBT) as parallel arrays of rows, columns, and values (RCV).
Class Size (lines)	1800 (with StandardRowSortedTable, StandardTable)
Public Methods	27
Raw Execution (seconds)	< 1.0
Alembic Script (lines)	33

	Daikon	**Alembic**
Number of Invariants	517	269
Trace file compressed (M)	0.054	0.122
Trace file uncompressed (M)	1.8	1.3
Execution Time (sec)	2	3

Listing 4.8. Fragment of Alembic file HBT.alembic that views HBT as an RCV trait.

```
trait RCV
[
    use com.google.common.collect.HashBasedTable<Integer,Integer,Integer>;
    state Object[] rows;
    state Object[] cols;
    state Object[] vals;
]

view ToRCV
[
    lift com.google.common.collect.HashBasedTable<Integer,Integer,Integer> to
        RCV;

    lift_state
    {
      Set<Table.Cell<Integer,Integer,Integer>> cset = (Set) this.cellSet();

      abs.rows = new Object[ cset.size() ];
      abs.cols = new Object[ cset.size() ];
      abs.vals = new Object[ cset.size() ];

      int i = 0;
      for ( Table.Cell<Integer,Integer,Integer> cell : cset )
      {
        abs.rows[ i ] = cell.getRowKey();
        abs.cols[ i ] = cell.getColumnKey();
        abs.vals[ i ] = cell.getValue();
        ++i;
      }
    }
]
```

Listing 4.9. Alembic invariants for the put and remove methods of HBT as exercised by HBTUnitTest using the view ToRCV to abstract the concrete object to the RCV trait.

```
RCV.put(java.lang.Object, java.lang.Object, java.lang.Object):::ENTER
 rowKey != null
 columnKey != null
 value != null

RCV.put(java.lang.Object, java.lang.Object, java.lang.Object):::EXIT
 size(this.rows[]) >= 1
 orig(rowKey) in this.rows[]
 orig(columnKey) in this.cols[]
 orig(value) in this.vals[]
 size(this.rows[]) >= orig(size(this.rows[]))
 size(this.rows[])−1 <= orig(size(this.rows[]))
 size(this.rows[])−1 >= orig(size(this.rows[]))−1

RCV.remove(java.lang.Object, java.lang.Object):::ENTER
 size(this.rows[]) one of { 2, 3 }

RCV.remove(java.lang.Object, java.lang.Object):::EXIT
 size(this.rows[]) one of { 1, 2, 3 }
 size(this.rows[]) <= orig(size(this.rows[]))
 size(this.rows[]) >= orig(size(this.rows[]))−1
 size(this.rows[])−1 <= orig(size(this.rows[]))−1
```

Listing 4.10. Daikon invariants for the put and remove methods of TBT exercised by TBTUnitTest.

```
StandardTable.put(java.lang.Object, java.lang.Object, java.lang.Object):::ENTER
  this.cellSet == null
  this.columnKeySet == null
  this.values == null
  this.rowMap == null
  rowKey != null
  columnKey != null
  value != null

StandardTable.put(java.lang.Object, java.lang.Object, java.lang.Object):::EXIT
  this.backingMap == orig(this.backingMap)
  this.factory == orig(this.factory)
  this.cellSet == orig(this.cellSet)
  this.rowKeySet == orig(this.rowKeySet)
  this.columnKeySet == orig(this.columnKeySet)
  this.values == orig(this.values)
  this.rowMap == orig(this.rowMap)
  this.columnMap == orig(this.columnMap)
  this.cellSet == null
  this.columnKeySet == null
  this.values == null
  this.rowMap == null

StandardTable.remove(java.lang.Object, java.lang.Object):::ENTER
  this.cellSet == null
  this.columnKeySet == null
  this.values == null
  this.rowMap == null

StandardTable.remove(java.lang.Object, java.lang.Object):::EXIT187
  this.cellSet == return
  this.rowKeySet == return
  this.columnKeySet == return
  this.values == return
  this.rowMap == return
  this.columnMap == return
  this.backingMap has only one value
  this.factory has only one value
  return == null
  orig(this) has only one value
```

We can use an identical Alembic abstraction to that we used in the previous section on HBT. We capture the table's cells into three parallel arrays corresponding to the *rows*, *columns* and *values* of the table; these arrays are more suited to Daikon's inference, as well as human understanding. We call the trait RCV (row, column, value), and place the trait and its view in an TBT.alembic file as in Listing 4.11 below; we don't show the program, execution and analysis because they are boilerplate text unrelated to abstraction.

Listing 4.11. Fragment of Alembic file TBT.alembic that views TBT as an RCV trait.

```
trait RCV
[
    use com.google.common.collect.TreeBasedTable <Integer,Integer,Integer>;
    state Object[] rows;
    state Object[] cols;
    state Object[] vals;
]

view ToRCV
[
    lift com.google.common.collect.TreeBasedTable<Integer,Integer,Integer> to
        RCV;

    lift_state
    {
      Set<Table.Cell<Integer,Integer,Integer>> cset = (Set) this.cellSet();

      abs.rows = new Object[ cset.size() ];
      abs.cols = new Object[ cset.size() ];
      abs.vals = new Object[ cset.size() ];

      int i = 0;
      for ( Table.Cell<Integer,Integer,Integer> cell : cset )
      {
        abs.rows[ i ] = cell.getRowKey();
        abs.cols[ i ] = cell.getColumnKey();
        abs.vals[ i ] = cell.getValue();
        ++i;
      }
    }
]
```

In Listing 4.12, we show the put and remove output from the Alembic run, as it is representative of the type of information available via our RCV trait.

Listing 4.12. Alembic invariants for the put and remove methods of TBT as exercised by TBTUnitTest using the view ToRCV to abstract the concrete object to the RCV trait.

```
RCV.put(java.lang.Object, java.lang.Object, java.lang.Object):::ENTER
    rowKey != null
    columnKey != null
    value != null

RCV.put(java.lang.Object, java.lang.Object, java.lang.Object):::EXIT
    size(this.rows[]) >= 1
    orig(rowKey) in this.rows[]
    orig(columnKey) in this.cols[]
    orig(value) in this.vals[]
    size(this.rows[]) >= orig(size(this.rows[]))
    size(this.rows[])−1 <= orig(size(this.rows[]))
    size(this.rows[])−1 >= orig(size(this.rows[]))−1

RCV.remove(java.lang.Object, java.lang.Object):::ENTER
    size(this.rows[]) one of { 2, 3 }

RCV.remove(java.lang.Object, java.lang.Object):::EXIT
    size(this.rows[]) one of { 1, 2, 3 }
    size(this.rows[]) <= orig(size(this.rows[]))
    size(this.rows[]) >= orig(size(this.rows[]))−1
    size(this.rows[])−1 <= orig(size(this.rows[]))−1
```

4.2.4. HashMultiset

The HashMultiset (HM) class represents a multiset using a java.util.HashMap as its storage. We choose to abstract this data structure as an array consisting of the distinct values from the underlying HM; in other words, as a Set. We summarize the comparison between Daikon and Alembic for this experiment Table 4.4, and briefly discuss the comparison below. In the same way that HBT and TBT used hidden superclasses as their primary implementation, the HM class uses hidden superclasses AbstractMapBasedMultiset and AbstractMultiset for the bulk of its implementation; we include these superclasses in the source line count, as before.

Table 4.4. Alembic and Daikon analyses of HM.

Abstraction	HashMultiset (HM) as a Set.
Class Size (lines)	730 (with AbstractMapBasedMultiset, AbstractMultiset)
Public Methods	20
Raw Execution (seconds)	< 1.0
Alembic Script (lines)	18

	Daikon	**Alembic**
Number of Invariants	222	165
Trace file compressed (M)	0.047	0.043
Trace file uncompressed (M)	0.963	0.504
Execution Time (sec)	2	2

In Listing 4.15, we show the add and remove output from the Alembic run, as it is representative of the type of information available via our RCV trait.

4.2.5. ArrayListMultimap

The ArrayListMultimap (ALM) class represents a multimap using instances of java.util.ArrayList as its underlying storage. We choose to abstract this data structure as an array consisting of only the keys from the underlying ALM. We summarize the comparison between Daikon and Alembic for this experiment Table 4.5, and briefly discuss the comparison below. In the same way that HBT and TBT used hidden superclasses as their primary implementation, the ALM class uses hidden superclasses AbstractMapBasedMultiset and AbstractMultiset for the bulk of its implementation; we

Listing 4.13. Daikon invariants for the add and remove methods of HM exercised by HMUnitTest.

```
AbstractMultiset.add(java.lang.Object):::ENTER
  this.elementSet == null
  this.entrySet != null
  element != null

AbstractMultiset.add(java.lang.Object):::EXIT
  this.elementSet == orig(this.elementSet)
  this.entrySet == orig(this.entrySet)
  this.elementSet == null
  this.entrySet != null
  return == true

AbstractMultiset.remove(java.lang.Object):::ENTER
  this.elementSet == null

AbstractMultiset.remove(java.lang.Object):::EXIT
  this.elementSet == orig(this.elementSet)
  this.entrySet == orig(this.entrySet)
  (return == false) ==> (orig(element) has only one value)
  (return == false) ==> (orig(this) has only one value)
  (return == false) ==> (this.entrySet has only one value)
  this.elementSet == null
```

Listing 4.14. Fragment of Alembic file HM.alembic that views HM as a Set trait.

```
trait Set
[
  use com.google.common.collect.HashMultiset<Object>;
  state Object[] elements;
]

view ToSet
[
  lift com.google.common.collect.HashMultiset<Object> to Set;

  lift_state
  {
    Set<Object> cset = (Set) this.elementSet();
    abs.elements = cset.toArray();
  }
]
```

Listing 4.15. Alembic invariants for the add and remove methods of HM as exercised by HMUnitTest using the view ToRCV to abstract the concrete object to the RCV trait.

```
Set.add(java.lang.Object):::ENTER
  this.elements[] elements != null
  element != null

Set.add(java.lang.Object):::EXIT
  this.elements[] elements != null
  return == true
  orig(element) in this.elements[]
  size(this.elements[]) >= orig(size(this.elements[]))
  size(this.elements[])−1 <= orig(size(this.elements[]))
  size(this.elements[])−1 >= orig(size(this.elements[]))−1

Set.remove(java.lang.Object):::ENTER
  size(this.elements[]) one of { 0, 1, 2 }

Set.remove(java.lang.Object):::EXIT
  (return == false) ==> (orig(element) has only one value)
  (return == false) ==> (orig(this.elements) has only one value)
  (return == false) ==> (orig(this.elements[]) == [])
  (return == false) ==> (this.elements has only one value)
  (return == false) ==> (this.elements[] == [])
  (return == false) ==> (this.elements[] == orig(this.elements[]))
  (return == true) <==> (orig(size(this.elements[])) one of { 1, 2 })
  (return == true) ==> (orig(element) in orig(this.elements[]))
  (return == true) ==> (size(this.elements[]) one of { 0, 1, 2 })
  size(this.elements[]) one of { 0, 1, 2 }
  size(this.elements[]) <= orig(size(this.elements[]))
  size(this.elements[]) >= orig(size(this.elements[]))−1
  size(this.elements[])−1 <= orig(size(this.elements[]))−1
```

include these superclasses in the source line count, as before. In Listing 4.16 and Listing 4.17, we show the put and remove output from the Daikon run, and in Listing 4.19, we show the output from the Alembic run for the same two representative methods.

Table 4.5. Alembic and Daikon analyses of ALM.

Abstraction	ArrayListMultimap (ALM) as an array of the keys in the concrete object.
Class Size (lines)	1674 (with AbstractListMultimap, AbstractMultimap)
Public Methods	22
Raw Execution (seconds)	< 1.0
Alembic Script (lines)	18

	Daikon	**Alembic**
Number of Invariants	681	142
Trace file compressed (M)	1.3	0.841
Trace file uncompressed (M)	42	11
Execution Time (sec)	8	5

4.2.6. Summary of Guava Results

Although the Guava collection library consists of many classes, we have selected a representative from each of the main container types and presented possible abstractions for each. These abstractions that we chose are not necessarily the best for all purposes; however, Alembic is designed to make it easy to try out new abstractions in a convenient way, encouraging the exploration of abstraction possibilities. We present a summary of the experiments we performed and their associated metrics in Table 4.6.

One of Alembic's primary benefits is reduction in the amount of invariants reported. We can interpret the data in Table 4.6 in a more meaningful way by looking at a graph comparing the invariants reported by the two systems (Daikon and Alembic), as in (Figure 4.1). It is clear that the number of invariants reported by Alembic is consistently, and sometimes dramatically, lower than the corresponding Daikon number.

Listing 4.16. Daikon invariants for the put and remove methods of ALM exercised by ALMUnitTest (1 of 2).

```
AbstractMultimap.put(java.lang.Object, java.lang.Object):::ENTER
  this.map != null
  this.keySet == null
  this.multiset == null
  this.valuesCollection == null

AbstractMultimap.put(java.lang.Object, java.lang.Object):::EXIT
  this.map == orig(this.map)
  this.keySet == orig(this.keySet)
  this.multiset == orig(this.multiset)
  this.valuesCollection == orig(this.valuesCollection)
  this.entries == orig(this.entries)
  this.asMap == orig(this.asMap)
  this.map != null
  this.totalSize >= 1
  this.keySet == null
  this.multiset == null
  this.valuesCollection == null
  return == true
  this.totalSize − orig(this.totalSize) − 1 == 0
  AbstractMultimap.serialVersionUID > orig(this.totalSize)

AbstractMultimap.remove(java.lang.Object, java.lang.Object):::ENTER
  this.map != null
  this.totalSize >= 1
  this.keySet == null
  this.multiset == null
  this.valuesCollection == null
```

Listing 4.17. Daikon invariants for the put and remove methods of ALM exercised by ALMUnitTest (2 of 2).

```
AbstractMultimap.remove(java.lang.Object, java.lang.Object):::EXIT
  this.map == orig(this.map)
  this.keySet == orig(this.keySet)
  this.multiset == orig(this.multiset)
  this.valuesCollection == orig(this.valuesCollection)
  this.entries == orig(this.entries)
  this.asMap == orig(this.asMap)
  (orig(key) has only one value) ==> (orig(this) has only one value)
  (orig(key) has only one value) ==> (orig(this.totalSize) == 1)
  (orig(key) has only one value) ==> (orig(value) has only one value)
  (orig(key) has only one value) ==> (return == false)
  (orig(key) has only one value) ==> (this.asMap has only one value)
  (orig(key) has only one value) ==> (this.entries has only one value)
  (orig(key) has only one value) ==> (this.map has only one value)
  (orig(key) has only one value) ==> (this.totalSize == 1)
  (orig(key) has only one value) ==> (this.totalSize == orig(this.totalSize))
  (return == false) <==> (this.totalSize == orig(this.totalSize))
  (return == false) ==> (orig(this.totalSize) one of { 1, 2 })
  (return == false) ==> (this.totalSize one of { 1, 2 })
  (return == true) ==> (this.totalSize − orig(this.totalSize) + 1 == 0)
  this.map != null
  this.keySet == null
  this.multiset == null
  this.valuesCollection == null
  this.totalSize <= orig(this.totalSize)
  AbstractMultimap.serialVersionUID > orig(this.totalSize)
```

Listing 4.18. Fragment of Alembic file ALM.alembic that views ALM as a Set trait.

```
trait Set
[
  use com.google.common.collect.ArrayListMultimap<Object>;
  state Object[] elements;
]

view ToKeys
[
  lift com.google.common.collect.ArrayListMultimap<Object,Object,Object> to Keys;
  lift_state
  {
    Set<Object> keyset = (Set) this.keySet();
    abs.keys = keyset.toArray();
  }
]
```

Listing 4.19. Alembic invariants for the put and remove methods of ALM as exercised by ALMUnitTest using the view ToRCV to abstract the concrete object to the RCV trait.

```
Keys.put(java.lang.Object, java.lang.Object):::ENTER

Keys.put(java.lang.Object, java.lang.Object):::EXIT
  return == true
  size(this.keys[]) >= 1
  orig(key) in this.keys[]
  size(this.keys[]) >= orig(size(this.keys[]))
  size(this.keys[])−1 <= orig(size(this.keys[]))

Keys.remove(java.lang.Object, java.lang.Object):::ENTER
  size(this.keys[]) one of { 1, 2 }

Keys.remove(java.lang.Object, java.lang.Object):::EXIT
  (return == false) ==> (size(this.keys[]) one of { 1, 2 })
  (return == false) ==> (this.keys[] == orig(this.keys[]))
  (return == true) ==> (orig(key) in orig(this.keys[]))
  (return == true) ==> (size(this.keys[]) one of { 0, 1, 2 })
  size(this.keys[]) one of { 0, 1, 2 }
  size(this.keys[]) <= orig(size(this.keys[]))
  size(this.keys[]) >= orig(size(this.keys[]))−1
  size(this.keys[])−1 <= orig(size(this.keys[]))−1
```

Table 4.6. Guava collection classes and simple Alembic abstractions.

Class *Abstracted to*	**Lines**	**Public Methods**	**Daikon Invs**	**Alembic Script**	**Alembic Invs**
MinMaxPriorityQueue	947	16	352	18	34
Sorted array of Integer					
Notes: Daikon produces a large number of invariants relating variables internal to the implementation. Figures 4.5 and 4.3 compare raw Daikon invariants to Alembic invariants.					
HashBasedTable	1350	22	434	33	253
Parallel arrays of row keys, column keys, and values					
Notes: Public interface methods for HashBasedTable are mostly synthesized "bridge" methods to a package-private superclass, StandardTable. Alembic reports the invariants for public methods in the abstraction, regardless of where they are implemented. Daikon reports almost nothing for HashBasedTable (it is blind to synthesized bridge methods), but reports corresponding invariants for like-named methods in implementing classes. The counts here include invariants for those implementing methods.					
TreeBasedTable	1800	27	517	33	269
Parallel arrays of row keys, column keys, and values					
Notes: As above, Daikon reports invariants for methods in package-private Guava superclasses (StandardRowSortedTable and StandardTable) that implement the functionality of TreeBasedTable via bridge methods in the public interface.					
HashMultiset	730	20	222	18	165
Array of distinct elements (i.e., as a Set)					
Notes: As above, Daikon reports invariants for methods in package-private Guava superclasses (AbstractMapBasedMultiset and AbstractMultiset) that implement the functionality of HashMultiset via bridge methods in the public interface.					
ArrayListMultimap	1674	22	681	18	142
Array of keys					
Notes: As above, Daikon reports invariants for methods in package-private Guava superclasses (AbstractListMultimap and AbstractMultimap) that implement the functionality of ArrayListMultimap via bridge methods in the public interface.					

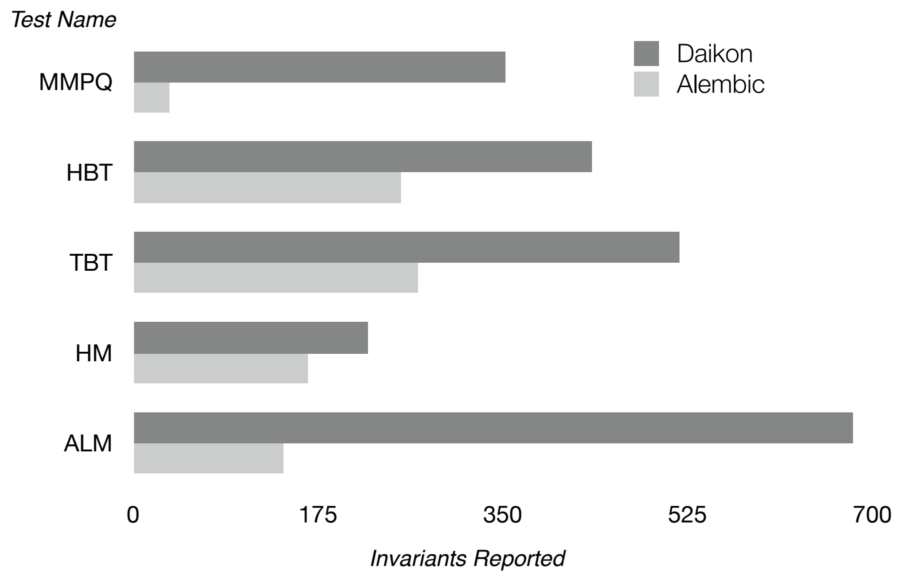


Figure 4.1. Graph showing reported invariants of Daikon and Alembic when performing the various Guava experiments in this section.

4.3. Proofs of Concept

Alembic was initially designed as a way to conveniently perform state abstraction experiments, with the intent of looking at various specialized abstraction concepts and techniques, and demonstrating their potential utility with Alembic. These specialized techniques could then be considered as potential features to be built in to future implementations of invariant detectors. In that vein, we'd like to present some of the more exotic forms of abstraction we have explored with Alembic, and suggest research problems and directions associated with them. These examples will not be as rigorous in terms of metrics as the Guava examples above; in many cases, there is no analogous feature in Daikon with which to compare our Alembic results. We invite the reader to consider possible extrapolations of these basic abstraction patterns.

4.3.1. Aggregate Abstraction and History Constraints

Given a module where can we can partition its methods into different logical groups (e.g., readonly methods, insertion methods, deletion methods), we can define an abstraction of the module that contains a function for each of these groups. By abstracting each concrete method into the shared method corresponding to its group, we enable an invariant detector to infer the pre and post conditions common to each member of a group. This is another example of aggregate abstraction, which we refer to in 2.6.2. Inferring Behavioral Subtypes and 2.6.4. Variable Hierarchy.

Aggregate abstraction is a technique where we take tuples originating from more than one concrete program point and merge them into a single set; we can associate such merged sets with the entry and exit of an abstract function (i.e., a method declared in a trait), and the resulting invariants will summarize the behavior of the source program points. We have utilized this general technique in deriving *history constraints* as described in [Liskov and Wing, 1994] and [Leavens, 2006]. A history constraint is a set of precondition and postconditions that must hold for any public method of a class or any of its subclasses.

We can compute an aggregate precondition and postcondition in Alembic by using a *many-to-one* projection of the concrete class's instance methods onto a single builtin trait method called invoke. The signature for this method is void invoke(); it takes no arguments and returns no result. However, it does provide program points for the back-end invariant detector to perform detection. When an Alembic view specifies a target method in the trait, a many-to-one projection is implied, and Alembic ensures that traces are propagated from the concrete entry to the aggregate method's entry; similarly, traces from the concrete exits are propagated to the aggregate method's exit. Because this aggregate method has no arguments, only the abstracted state is observable in the trait. The resulting invariant detection upon the aggregate method will reveal the generalized history constraint for those concrete methods observed. From Liskov and Wing [Liskov and Wing, 1994], the *history constraint* is part of a type specification that contains a set of preconditions and postconditions that must hold for any public method the type, or any of its subtypes.

To make this clearer, we provide an example below. The example will be based upon the simple Container class in Listing 4.20 and the test driver in Listing 4.21. We will use the Alembic file in Listing 4.22 to capture two traits, Length and LengthHistory. The first, Length, will be an ordinary state abstraction, where each concrete method results in the synthetic invocation on a corresponding method in the trait. For example, Container.add(5) will result in an invocation of Length.add(5). We expect to see invariants reported on the Length trait for each of the add, addTwo and addAll methods, and this is indeed the case, as shown in the first part of Listing 4.23.

The second trait we derive, LengthHistory, is populated via the ToLengthHistory view, which performs the same state abstraction and instruments the same concrete Container methods. However, it lifts each of the concrete methods into a single, zero-argument, method called invoke() on the trait. The resulting invariants on LengthHistory.invoke shown in the second part of Listing 4.23 reveal the common behavior of each of the add* methods in terms of their precondition and postcondition on the abstracted aggregate method.

Listing 4.20. The Container Java class.

```
import java.util.Arrays;
import java.util.Collections;

public class Container
{
    private Object opaqueArray[];

    public Container()
    {
        reset();
    }

    public void reset()
    {
        opaqueArray = new Object[ 0 ];
    }

    public void add( Object element )
    {
        int oldLen = opaqueArray.length;
        opaqueArray = Arrays.copyOf( opaqueArray, oldLen + 1 );
        opaqueArray[ oldLen ] = element;
    }

    public void addTwo( Object element1, Object element2 )
    {
        int oldLen = opaqueArray.length;
        opaqueArray = Arrays.copyOf( opaqueArray, oldLen + 2 );
        opaqueArray[ oldLen ] = element1;
        opaqueArray[ oldLen + 1 ] = element2;
    }

    public void addAll( Object[] elements )
    {
        int oldLen = opaqueArray.length;
        opaqueArray = Arrays.copyOf( opaqueArray, oldLen + elements.length );
        System.arraycopy( elements, 0, opaqueArray, oldLen, elements.length );
    }
}
```

Listing 4.21. The ContainerTester Java class.

```
import java.util.*;

public class ContainerTester
{
  public static void main(String[] args)
  {
    int outerBound = 3;
    int bound = 10;
    Container c = new Container();

    for ( int i = -outerBound; i <= outerBound; ++i )
    {
        c.reset();

        Object[] os = new Object[ bound ];
        for ( int j = 1; j <= bound; ++j )
        {
            c.add( i + j );
            os[ j - 1 ] = i * j;
            c.addTwo( i, j );
        }

        c.addAll( os );
    }
  }
}
```

Listing 4.22. The ContainerHistory.alembic example uses aggregate abstraction to summarize the common behavior of add* methods.

```
trait Length
[
    path "../../examples/Container/";
    use Container : "^add";
    state int length;
]

trait LengthHistory
[
    state int length;
]

view ToLengthT
[
    path "../../examples/Container/";
    lift_state
    {
        abs.length = opaqueArray.length;
    }
]

view ToLength : ToLengthT
[
    lift Container : "^add" to Length;
]

view ToLengthHistory : ToLengthT
[
    lift Container : "^add" to LengthHistory.invoke;
]
```

Listing 4.23. The Alembic invariants for the ContainerHistory.alembic example.

```
Length:::OBJECT
  this has only one value

Length.add(java.lang.Object):::ENTER
  element != null

Length.add(java.lang.Object):::EXIT
  this.length − orig(this.length) − 1 == 0

Length.addAll(java.lang.Object[]):::ENTER
  this.length == 30
  size(elements[]) == 10

Length.addAll(java.lang.Object[]):::EXIT
  elements[] == orig(elements[])
  this.length == 40

Length.addTwo(java.lang.Object, java.lang.Object):::ENTER
  element1 != null
  element2 != null

Length.addTwo(java.lang.Object, java.lang.Object):::EXIT
  this.length − orig(this.length) − 2 == 0


LengthHistory:::OBJECT
  this has only one value

LengthHistory.invoke():::ENTER

LengthHistory.invoke():::EXIT
  this.length > orig(this.length)
```

4.3.2. Abstract Functions and Loop Abstraction

In the examples below, we look at a class of abstraction techniques where the methods in the abstracted trait do not correspond to concrete method invocations at all. Rather, they correspond to an *abstract function* from an abstract entry point to an abstract exit point. These points may correspond to specific code locations, as in the case of Loop Abstraction below. Or they may correspond to the occurrence of more abstract events, such as the beginning and end of a timer. However, the mechanism we use to synthesize these abstract functions is unrestricted, and we can conceive of potential applications of this, some of which we describe in 5.1. Future Work.

In the discussion below, we sometimes use the term *arrow* to refer to these abstract functions to emphasize that they have no implementation and do not correspond to existing functions or methods. An arrow is defined by the sets of tuples corresponding to its entry (the domain) and its exit (the codomain). An invariant detector is able to describe such an arrow with invariants.

The composition of two or more functions may be treated as a method on an abstraction object, where the entry conditions for the first function become the entry conditions of the abstract method, and the exit conditions for the composition become the exit conditions for the abstract method. Note that this mechanism can be easily applied to a procedural language such as Java by considering each statement to be a function from old state to new state, and by considering a sequence of statements to be a composition of functions. In other words, we can treat a sequence of statements as an composition of functions, and we can consider the abstract arrow from the beginning of this sequence to its end.

What Alembic provides is a way to abstract over these compositions or sequences, and to produce invariants that describe the change induced by them. In fact, the primary capability of any invariant detector is to infer, for each targeted method, an abstract arrow over the fixed sequence of statements comprising that method's implementation. Alembic enables this invariant detection capability to be applied anywhere we can call Alembic's liftEntry and liftExit methods to inject traces into an arrow. We show a practical use for this technique by using Alembic to infer loop properties in the examples below.

Loop Abstraction is actually a special application of *abstract functions*, where we view one part of the loop structure as an entry point, and another part as an exit point for the composition, and the loop exit as the end. Applying an invariant detector to the resulting abstract function or arrow enables the discovery of loop properties that might be unobtainable otherwise. It is necessary to insert explicit calls to Alembic functions in order to use abstract functions; we did not need to insert explicit calls in our prior Guava examples of simple state abstraction, because the concrete method entry and exit can be instrumented by AspectJ to insert these calls on behalf of the user.

We will provide two examples of how Alembic can be used to capture loop properties. The first will demonstrate the creation of an abstract function corresponding to the per-iteration body of the loop and of an abstract function corresponding to the loop as a whole. These two arrows will be specified as methods on the LoopArrows trait defined in Listing 4.26. The second example will compute the loop invariant as the state of the LoopInvariants abstraction object defined in the same file. In both examples, we will synthesize traces to our arrows (abstract functions) by inserting probe pairs into the source of the Java function in Listing 4.24 and Listing 4.25. Each pair of probes corresponds to the head and tail of an abstract function arrow from the visible state at the head probe to the visible state at the tail probe.

Listing 4.24. Two pairs of Alembic probes added to a loop that sums the first n odd numbers to compute n^2. The first pair synthesizes an invocation upon the LBLA (LoopBeforeLoopAfter) method of the LoopArrows trait; the second synthesizes an invocation upon the BBBA (BodyBeforeBodyAfter) method.

```
static void runLoopArrows( int bound )
{
    Abstraction_LoopArrows loop = Abstraction_LoopArrows.getTheAbstraction();
    for ( int n = 1; n < bound; ++n )
    {
        int sum = 0;
        int i = 0;

        Alembic.InvokeContext lblaContext = loop.liftEntry_LBLA( n );

        while ( i < n )
        {
            Alembic.InvokeContext bbbaContext = loop.liftEntry_BBBA( n, i, sum );

            i = i + 1;
            sum += ( i − 1 ) ∗ 2 + 1; // Add next odd number

            loop.liftExit_BBBA( bbbaContext, n, i, sum, sum );
        }

        loop.liftExit_LBLA( lblaContext, n, sum );
    }
}
```

Listing 4.25. Explicit probes capture the loop invariant as the object invariant of the LoopInvariants trait.

```
static void runLoopInvariants( int bound )
{
    Abstraction_LoopInvariants loop = Abstraction_LoopInvariants.getTheAbstraction
        ();

    for ( int n = 1; n < bound; ++n )
    {
        int sum = 0;
        int i = 0;

        loop.sum = sum;
        loop.i = 0;
        Alembic.InvokeContext probeContext = loop.liftEntry_probe();

        while ( i < n )
        {
            i = i + 1;
            sum += ( i − 1 ) ∗ 2 + 1;

            loop.sum = sum;
            loop.i = i;
            loop.liftExit_probe( probeContext, null ); // end one arrow
            probeContext = loop.liftEntry_probe(); // begin the next
        }

        loop.sum = sum;
        loop.i = i;
        loop.liftExit_probe( probeContext, null ); // finish the last arrow
    }
}
```

In the Loops.alembic file (Listing 4.26), we define two traits, LoopArrows and LoopInvariants. The first of these contains methods corresponding to the following functional arrow types:

LoopBeforeLoopAfter (LBLA) This method treats the entire loop execution as a function from the loop's pre-state to the loop's post-state. Graphically, we are creating an abstract function that is the arrow from before a loop to after the loop. This treats the entire loop as a *black box* and derives a description of its behavior.

BodyBeforeBodyAfter (BBBA) This method looks at the inner body of the loop as though it were a function from the values of variables at the top of the loop body to the values at the bottom of the loop body. Graphically, we are creating the arrow from the top of the loop body to the bottom of the loop body. This should produce a description that characterizes an arbitrary execution of the loop body.

Listing 4.26. Loops.alembic defines the LoopArrows and LoopInvariants traits.

```
trait LoopArrows
[
    method "int LBLA( int n )";
    method "int BBBA( int n, int i, int oldSum )";
]

trait LoopInvariants
[
    state int i;
    state int sum;

    method "void probe()";
]

program LoopP
[
    builddir "../../examples/loops/";
    traits "LoopArrows LoopInvariants";
]
```

The Alembic build process will generate code for both traits and will then invoke Daikon to execute the program, resulting in the generation of execution traces and ultimately, invariants. If we examine Listing 4.27 we see descriptions of two traits. Notice how LBLA(int):::EXIT characterizes the black box behavior of the loop, and how LoopInvariants:::OBJECT contains the loop invariants for our loop example. In each case, the return variable corresponds to the result we fed to the probe at the end of our respective arrow.

Because loop invariant detection involves treating a subregion of a function as though it were a composition of functions, it is necessary to delimit this subregion somehow. In the current prototype of Alembic, this is done by the explicit insertion of liftEntry and liftExit calls and state abstraction into the target code. We anticipate that a future version of Alembic could use a more convenient mechanism (e.g., the Java 1.5 *annotations* feature) for delimiting the subregion that comprises a loop. This *syntactic sugar* would still require

Listing 4.27. The Alembic invariants for LoopArrows.

```
LoopArrows.BBBA(int, int, int):::ENTER
  i >= 0
  oldSum >= 0
  n > i
  i <= oldSum
  oldSum == i**2

LoopArrows.BBBA(int, int, int):::EXIT
  return >= 1
  return > orig(i)
  return > orig(oldSum)
  return − 2 * orig(i) − orig(oldSum) − 1 == 0

LoopArrows.LBLA(int):::EXIT
  return % orig(n) == 0
  return == orig(n)**2
  return >= orig(n)

LoopInvariants:::OBJECT
  this has only one value
  this.i >= 0
  this.sum >= 0
  this.i <= this.sum
  this.sum == this.i**2
```

a slight modification to the target code in the form of a call to some placeholder functions Alembic.BeginArrow and Alembic.EndArrow function; these would then be detected by an AspectJ aspect that would transform it into the form desired by Alembic.

CHAPTER V

CONCLUSION

We complete this dissertation by summarizing the ideas and results presented, looking at some of the directions this research might take, and concluding with some remarks and recommendations regarding dynamic invariant detection and abstraction.

5.1. Future Work

The current implementation of Alembic has proven sufficient to demonstrate a variety of abstraction techniques and applications. There are several promising directions for both the Alembic technology and the theory and applications of abstracted dynamic invariant detection. We sketch some of these ideas below, including how they might be implemented.

5.1.1. Method Transformation

Although we haven't written any examples at this time, Alembic has a powerful feature which allows a user to control the per-method abstraction in a fine-grained way. Alembic provides an optional lift_method clause in the view declaration syntax; this clause is similar to the lift_state clause in that it provides a way for Java code to be inserted. However, the lift_method clause has an entry and exit handler, where the user may specify custom handling of the lift-on-entry and lift-on-exit behavior. We show an example of this in Listing 5.1, which is similar to the example of aggregate abstraction in 4.3.1. Aggregate Abstraction and History Constraints and uses the same base class Container and unit test ContainerTester (Listing 4.20 and Listing 4.21, respectively). For the sake of space in this dissertation, we only show the implementation of the addTwo method, although add and addAll are just as simple.

Unlike the previous Container example, we explicitly capture the arguments of each add* method and abstract them as an integer which is fed to increment on the aggregate method, addToSum(int increment). The resulting invariants on LengthHistorySum are displayed in Listing 5.2. The first two exit invariants reveal the underlying behavior; because we chose to define a trait with both a length and sum attribute, we get the inevitable irrelevant invariants, such as this.length != orig(this.sum). This could have been avoided if we had created separate traits; however, we wanted to demonstrate the power of the lift_method clause in terms of its ability to capture and transform both state and argument variables into new state and argument variables, perhaps even invoking an abstract function, as in this example.

An unexplored potential use of this lift_method feature of Alembic is that we can implement conditional abstractions, where the view and its associated lift_method clauses may control where to and whether to propagate a trace. The Csallner/Smaragdakis paper on behavioral subtyping relies upon this conditional propagation.

Other cases where conditional abstraction may be useful include:

- Dynamic filtering of traces prior to invariant detection
- Case-splitting of traces into multiple abstractions (the inverse of aggregate abstraction)

The current implementation of lift_method is very *raw* in the sense that the entirety of the lifting process is laid bare for an analyst to use, and we hope that this power finds a good use. It does require that the analyst create potentially tedious lift_method clauses for each method, however. We believe that the same *wildcard* feature provided by Alembic's use and lift clauses can be extended to allow a template-based way to specify multiple lift_method clauses with a single directive. Example uses might be to transform concrete arguments into an abstraction-compatible form; a wildcard facility would allow this transformation to be specified once in a lift_method clause, and it would be applied to all the implicitly generated lifting code.

5.1.2. Supertype Abstraction

Our approach to abstraction was initially inspired by the Csallner and Smaragdakis work on inferring behavioral subtype specifications using dynamic invariant detection. Their approach used trace propagation to transform concrete subclass invocations to synthetic invocations on a superclass object, and then used a dynamic invariant detector to infer specifications about the superclass [Csallner and Smaragdakis, 2006]. We call this technique *supertype abstraction*, which is also the name used by Leavens to refer to the principle of modular reasoning that an object reference of a given superclass can be bound to an instance of that superclass or any of its subclasses [Leavens and Naumann, 2006]. Leavens proves that a type hierarchy organized as behavioral subtypes will satisfy the principle of supertype abstraction (see 2.6.2. Inferring Behavioral Subtypes).

One direction that could be taken with Alembic is to continue the work proposed by Csallner and Smaragdakis for inferring behavioral subtype specifications. A rough sketch of how this could be done is presented below. This would involve the development of additional tools to analyze the class hierarchy and generate intermediate files. We will

Listing 5.1. This ContainerHistorySum example defines a trait containing length and sum state variables, and a single method addToSum(int increment). The full declaration of add and addAll has been left out for space reasons..

```
trait LengthHistorySum [
  state int length;
  state int sum;
  method "void addToSum( int increment )";
]

view ToLengthHistorySum : ToLengthT [
  lift Container : "^add" to LengthHistorySum;

  lift_state {
    abs.length = opaqueArray.length;
  }

  lift_method "void addTwo(Object e1, Object e2)"
  [
    state int enterSum;
    entry {
        int i1 = (int) ((Integer) element1 );
        int i2 = (int) ((Integer) element2 );
        enterSum = i1 + i2;
        context = absThis.liftEntry_addToSum( enterSum );
    }

    exit {
        absThis.sum += enterSum;
        absThis.liftExit_addToSum( context, enterSum, concreteResult );
    }
  ]

  lift_method "void add(Object element)" [
        // Elided for brevity ]

  lift_method "void addAll(Object[] elements)" [
        // Elided for brevity ]
]
```

assume, for the outline below, that we have a class hierarchy consisting of a superclass S and its two subclasses L and R. The method .m() is defined in S and overridden in L.

Generate Abstraction Hierarchy For each class (S, L, R), generate an Alembic trait containing state variables corresponding to the instance variables on that class and containing method definitions corresponding to each of the instance methods on that class. Let's call these traits TraitS, TraitL, and TraitR.

Generate View Hierarchy For each class (S, L, R), generate an Alembic view that lifts invocations to the corresponding trait, as well as to traits corresponding to supertypes. For example, an invocation on L.m() should be lifted to invocations on TraitL.m() as well as TraitR.m(). This capability can be built upon the Alembic lift_method syntax, which allows explicit control over the trace propagation and transformation mechanism.

The Csallner and Smaragdakis work shows that determining behavioral subtypes is not as simple as just using trace propagation. They point out that invocations on a subclass that do not satisfy the superclass's precondition should not be propagated. But these preconditions themselves need to be discovered before this determination can be made. So they propose a two-phase solution where the first phase determines preconditions, and the second phase uses these to selectively propagate invocations to superclasses. Implementing this in Alembic would require some equivalent to this two-phase solution.

5.1.3. Improved Lifecycle Control

Currently, traits are implemented as shared singleton objects that are constructed at program startup; this is sufficient for single-threaded applications where the only state in the abstraction is derived from entry and exit, and where this state is not required to

Listing 5.2. Alembic invariants for ContainerHistorySum. The first two exit invariants are the useful ones; the other two exit invariants are noise which could have been avoided if we had separated the trait into two traits.

```
LengthHistorySum:::OBJECT
  this.length >= 0

LengthHistorySum.addToSum(int):::EXIT
  this.length >= orig(this.length)
  this.sum − orig(this.sum) − orig(increment) == 0
  this.length != orig(this.sum)
  this.sum != orig(this.length)
```

persist between invocations. In other words, each abstraction has no history other than that derived from the concrete object at entry and exit. This encompasses all of our simple state abstractions described in Chapter IV.

However, there are experiments we can conceive where we need to associate state with an abstraction and where this state is derived from the unique history of the concrete object, but where this history is not maintained by the concrete object. For example, if we desire to have a File abstraction maintain a history of the operations invoked upon it, then we need a unique abstraction instance corresponding to each unique concrete instance.

This will require modifying the instrumentation layer to construct these abstractions. We are confident that our current use of AspectJ will make this modification possible. The same effort required to track concrete instance constructions and mirror them with abstraction constructions will likely result in Alembic being able to reasonably abstract constructors and static methods, which it does not currently support.

5.1.4. Other Ideas

There are several ideas and interesting directions we considered during this research, but haven't explored fully. These are briefly summarized below.

Protocols and Temporal Abstraction One research path suggested by Alembic is to explore the mining of temporal specifications that describe the legitimate sequences of method calls for an object. We would provide a function to abstract a method invocation and its state into a *token* that would be fed to a state-machine miner which generated a set of *candidate* finite-state machines. At the same time, the invariant detector would be used to associate invariants with certain states. It may be possible to use these building blocks to develop temporal specification miners as in [Lorenzoli et al., 2008] and [Ammons et al., 2002]. Lamport provides a nice description of how to map behavioral specifications in precondition/postcondition form into state machine format [Lamport, 1989].

Effect Abstraction A dynamic invariant detector such as Daikon is well-suited to capturing invariants over common datatypes such as floats, integers, and strings, as well as over some container classes (arrays, lists, sets). When the behavior of a class is characterized by its external effects, then it is necessary to use abstraction to capture these effects in a form that is amenable to invariant analysis. We can construct abstractions whose state is derived from *effects* that would ordinarily be unobservable by an invariant detector. For example, a File object's read and write methods could be abstracted such that the effect on the file system is captured as observable state on the abstraction object. This in turn enables an invariant detector to perform inference on the read and write methods' effects. Other effects that can be captured are size, time, and even the number of instructions executed in a method. We believe that this type of abstraction may lead to the ability of using invariant detection to compute resource complexity measures as invariants.

Layered Abstraction It is conceivable that one might apply the abstraction technique to a Java class that was itself an abstraction (i.e., generated by Alembic from a trait). Traces from the concrete would be propagated to its abstraction (the primary abstraction), where the same mechanism (but a different view and target trait) could be applied to propagate traces from the primary abstraction to the secondary. We haven't come up with anything but contrived examples for this, but the theory indicates that it is legitimate. Usually one who wanted to perform two abstractions lift_1 and lift_2 would simply compose the abstraction functions and use a single abstraction $\text{lift}_2 \circ \text{lift}_1$, without any intermediate abstraction object. However, a complex abstraction tower might necessitate the use of multiple abstraction layers.

Context-dependent Instrumentation AspectJ provides us with a powerful way to determine what our calling context is. This could be utilized to ensure that methods were only abstracted when invoked from outside of a module, for example. The ability to capture the static calling context would enable Alembic to implement supertype abstraction, which requires that we propagate traces to an abstraction corresponding to the static receiver.

Alternative Back-ends Alembic currently uses Daikon as its general-purpose back-end invariant detector. The design of Alembic allows for a trait to be hooked up to alternative, specialized detectors. These detectors would see an abstracted trace stream and perform specialized inference upon it, producing invariants and other types of behavioral description. Examples of detectors that might be useful include a regular expression detector and a regression solver.

Concretization Concretization is the inverse of abstraction. Once invariants are discovered over an abstraction, it may be desirable to take these invariants and automatically translate them back into invariants on the concrete object. If the user provided an unlift clause that indicated how to translate an abstract invariant into a concrete invariant, then Alembic could emit concrete invariants derived from abstractions.

5.2. Suggestions for Detectors

Alembic is not designed to replace traditional detectors, although we hope that some of the features demonstrated by Alembic get incorporated *natively* into the next generation of detection technology. In addition to these capabilities, there are some features of detectors that would make implementation of Alembic and abstraction systems easier and more powerful to use. We briefly describe these below:

Preloaded Invariants The ability to preload *known* invariants would allow specifications from other sources (e.g., from JML annotations) or from prior detector runs to be used as input, so that the invariant detector could build upon these to discover invariants that supplement the preloaded invariants. Optionally, these preloaded invariants would be excluded from the final report.

Supported Third-Party API Our development of Alembic required some minor modifications to Daikon so that we can synthesize traces. In addition, we ended up calling functions that were likely not intended for a layered facility like Alembic. If Daikon provided these functions (or API wrappers) as a documented and supported capability, it would make building facilities like Alembic easier and more robust.

5.3. Summary

We have described limitations in the application of Daikon-style dynamic invariant detection to concrete implementations. Specifically, the complexity of invariant expressions is limited due to memory and performance considerations, and the breadth and scope of invariant expressions are limited due to the use of a general-purpose vocabulary of invariant operators and terms. This results in output that is less useful due to the presence of true, but irrelevant invariants and due to true invariants being unreported because the requisite expressions are inexpressible in the general purpose and limited-depth vocabulary.

We described a new approach that addresses these limitations as well as providing several new capabilities. We showed that by creating abstractions of the concrete implementations, we can focus the invariant detector on arbitrarily deep and complex invariants, while filtering out extraneous variables and operators that would normally detract from the invariant output with the noise of true, but irrelevant, invariants.

We presented a new program analysis tool called Alembic that simplifies the expression of these abstractions and automates the requisite instrumentation and code generation. We used Alembic programs to illustrate a diverse set of applications of this mechanism, including history constraint detection, effect abstraction, loop invariant detection, and ad hoc abstraction of concrete implementations.

We can use the lessons learned here to inspire and inform the next generation of detectors. Abstraction is a practical tool for eliciting information via dynamic invariant detection, and the Alembic system is an extensible foundation for performing dynamic invariant detection experiments and applying new abstraction techniques.

APPENDIX A

SCRIPT TO RUN DAIKON ON MMPQ UNIT TEST

Listing A.1. Shell commands required to analyze MMPQUnitTest by first instrumenting with Chicory, then executing it against Guava's MMPQ implementation, and finally analyzing the results with Daikon.

```
export CLASSPATH="${CLASSPATH}:${GUAVA_JAR}:${GUAVA_TESTS}:${GUAVA_TESTLIB}:
    ${JUNIT_JAR}:${TRUTH_JAR}"
> javac -g MMPQUnitTest.java
> export MMPQ="^com.google.common.collect.MinMaxPriorityQueue"
> java \
      daikon.Chicory \
      --ppt-select-pattern='com.google.common.collect.[^.]+.offer' \
    --ppt-omit-pattern='^${MMPQ}.create' \
    --ppt-omit-pattern='^${MMPQ}.create' \
    --ppt-omit-pattern='^${MMPQ}.orderedBy' \
    --ppt-omit-pattern='^${MMPQ}.expectedSize' \
    --ppt-omit-pattern='^${MMPQ}.maximumSize' \
    --ppt-omit-pattern='^${MMPQ}.<init>' \
    --ppt-omit-pattern='^${MMPQ}.elementData' \
    --ppt-omit-pattern='^${MMPQ}.getMaxElementIndex' \
    --ppt-omit-pattern='^${MMPQ}.removeAt' \
    --ppt-omit-pattern='^${MMPQ}.fillHole' \
    --ppt-omit-pattern='^${MMPQ}.removeAndGet' \
    --ppt-omit-pattern='^${MMPQ}.heapForIndex' \
    --ppt-omit-pattern='^${MMPQ}.isEvenLevel' \
    --ppt-omit-pattern='^${MMPQ}.isIntact' \
    --ppt-omit-pattern='^${MMPQ}.capacity' \
    --ppt-omit-pattern='^${MMPQ}.initialQueueSize' \
    --ppt-omit-pattern='^${MMPQ}.growIfNeeded' \
    --ppt-omit-pattern='^${MMPQ}.calculateNewCapacity' \
    --ppt-omit-pattern='^${MMPQ}.capAtMaximumSize' \
    --ppt-omit-pattern='^${MMPQ}.<init>' \
    --ppt-omit-pattern='^${MMPQ}.access$500' \
    --ppt-omit-pattern='^${MMPQ}.access$600' \
    --ppt-omit-pattern='^${MMPQ}.access$700' \
    --omit-var='getClass' \
    --dtrace-file=MMPQUnitTest.dtrace.gz \
      MMPQUnitTest
> java \
      daikon.Daikon \
          --var-omit-pattern='getClass' \
          --config_option daikon.Daikon.progress_delay=-1 \
      MMPQUnitTest.dtrace.gz
```

APPENDIX B

FULL DAIKON RESULTS FOR MMPQ UNIT TEST

The listings Listing B.1 through Listing B.10 comprise the Daikon output when analyzing the MMPQ class with the Guava unit tests, as described in 4.2.1. MinMaxPriorityQueue. As before, we have shortened the name com.google.common.collect.MinMaxPriorityQueue to MMPQ for formatting purposes. The getClass() method has been excluded from the invariant detection process for both Alembic and Daikon.

Listing B.1. Daikon output for MMPQ (1 of 10).

```
MMPQ:::OBJECT
  this.minHeap != null
  this.maxHeap != null
  this.maximumSize one of { 42, 2147483647 }
  this.queue != null
  this.size >= 0
  this.modCount >= 0
  this.maximumSize > this.size
  this.maximumSize > this.modCount
  this.maximumSize != MMPQ.EVEN_POWERS_OF_TWO
  this.maximumSize > MMPQ.ODD_POWERS_OF_TWO
  this.maximumSize > MMPQ.DEFAULT_CAPACITY
  this.maximumSize > size(this.queue[])
  this.size <= this.modCount
  this.size < MMPQ.EVEN_POWERS_OF_TWO
  this.size > MMPQ.ODD_POWERS_OF_TWO
  this.size <= size(this.queue[])
  this.modCount < MMPQ.EVEN_POWERS_OF_TWO
  this.modCount > MMPQ.ODD_POWERS_OF_TWO
  MMPQ.EVEN_POWERS_OF_TWO > size(this.queue[])
  MMPQ.ODD_POWERS_OF_TWO < size(this.queue[])−1

MMPQ.add(java.lang.Object):::ENTER
  this.maximumSize == 2147483647
  element != null

MMPQ.add(java.lang.Object):::EXIT
  this.minHeap == orig(this.minHeap)
  this.maxHeap == orig(this.maxHeap)
  this.maximumSize == orig(this.maximumSize)
  this.queue[this.size−1] == this.queue[orig(this.size)]
  this.maximumSize == 2147483647
  this.size >= 1
  this.modCount >= 1
```

Listing B.2. Daikon output for MMPQ (2 of 10).

```
  return == true
  this.queue[this.size−1] != null
  this.maximumSize > orig(this.size)
  this.maximumSize > orig(this.modCount)
  this.maximumSize > orig(size(this.queue[]))
  orig(element) in this.queue[]
  this.size − orig(this.size) − 1 == 0
  this.size != orig(this.modCount)
  this.modCount > orig(this.size)
  this.modCount − orig(this.modCount) − 1 == 0
  MMPQ.EVEN_POWERS_OF_TWO > orig(this.size)
  MMPQ.EVEN_POWERS_OF_TWO > orig(this.modCount)
  MMPQ.ODD_POWERS_OF_TWO < orig(this.size)
  MMPQ.ODD_POWERS_OF_TWO < orig(this.modCount)
  orig(this.size) <= size(this.queue[])−1
  orig(this.modCount) != size(this.queue[])
  size(this.queue[]) >= orig(size(this.queue[]))
  size(this.queue[])−1 != orig(size(this.queue[]))
  size(this.queue[])−1 >= orig(size(this.queue[]))−1

MMPQ.addAll(java.util.Collection):::ENTER
  this.size == this.modCount
  MMPQ.DEFAULT_CAPACITY == size(this.queue[])
  this.queue[this.size] == this.queue[MMPQ.DEFAULT_CAPACITY−1]
  this.maximumSize == 2147483647
  this.queue[] contains only nulls and has only one value, of length 11
  this.queue[] elements == null
  this.size == 0
  this.queue[] elements == this.queue[this.size]

MMPQ.addAll(java.util.Collection):::EXIT
  this.minHeap == orig(this.minHeap)
  this.maxHeap == orig(this.maxHeap)
  this.maximumSize == orig(this.maximumSize)
  this.size == this.modCount
  this.queue[this.size] == orig(this.queue[post(MMPQ.DEFAULT_CAPACITY)−1])
  this.queue[this.size] == orig(this.queue[this.size])
  this.queue[this.size] == orig(this.queue[this.modCount])
  this.maximumSize == 2147483647
  return == true
  size(this.queue[]) one of { 11, 24, 50 }
  this.queue[this.size] == null
  this.size != MMPQ.DEFAULT_CAPACITY
  this.size > orig(this.size)
  this.size < size(this.queue[])−1
  this.size != orig(size(this.queue[]))−1
```

Listing B.3. Daikon output for MMPQ (3 of 10).

```
  MMPQ.DEFAULT_CAPACITY <= size(this.queue[])
  MMPQ.DEFAULT_CAPACITY != size(this.queue[])−1
  orig(this.queue[]) elements == this.queue[this.size]
  orig(this.size) < size(this.queue[])−1
  size(this.queue[])−1 >= orig(size(this.queue[]))−1

MMPQ.comparator():::ENTER
  this.size == this.modCount
  this.size one of { 0, 6 }
  size(this.queue[]) one of { 8, 11 }
  this.queue[this.size] == null
  this.size < MMPQ.DEFAULT_CAPACITY
  this.size < size(this.queue[])−1
  MMPQ.DEFAULT_CAPACITY >= size(this.queue[])

MMPQ.comparator():::EXIT
  this.minHeap == orig(this.minHeap)
  this.maxHeap == orig(this.maxHeap)
  this.maximumSize == orig(this.maximumSize)
  this.queue == orig(this.queue)
  this.queue[] == orig(this.queue[])
  this.size == this.modCount
  this.size == orig(this.size)
  this.size == orig(this.modCount)
  this.size one of { 0, 6 }
  return != null
  size(this.queue[]) one of { 8, 11 }
  this.queue[this.size] == null
  this.size < MMPQ.DEFAULT_CAPACITY
  this.size < size(this.queue[])−1
  MMPQ.DEFAULT_CAPACITY >= size(this.queue[])

MMPQ.iterator():::ENTER
  this.maximumSize == 2147483647
  size(this.queue[]) one of { 11, 24, 50 }
  this.queue[this.size] == null
  this.size != MMPQ.DEFAULT_CAPACITY
  this.size <= size(this.queue[])−1
  MMPQ.DEFAULT_CAPACITY <= size(this.queue[])
  MMPQ.DEFAULT_CAPACITY != size(this.queue[])−1
```

Listing B.4. Daikon output for MMPQ (4 of 10).

```
MMPQ.iterator():::EXIT
  this.minHeap == orig(this.minHeap)
  this.maxHeap == orig(this.maxHeap)
  this.maximumSize == orig(this.maximumSize)
  this.queue == orig(this.queue)
  this.queue[] == orig(this.queue[])
  this.size == orig(this.size)
  this.modCount == orig(this.modCount)
  this.maximumSize == 2147483647
  return != null
  size(this.queue[]) one of { 11, 24, 50 }
  this.queue[this.size] == null
  this.size != MMPQ.DEFAULT_CAPACITY
  this.size <= size(this.queue[])-1
  MMPQ.DEFAULT_CAPACITY <= size(this.queue[])
  MMPQ.DEFAULT_CAPACITY != size(this.queue[])-1

MMPQ.offer(java.lang.Object):::ENTER
  element != null

MMPQ.offer(java.lang.Object):::EXIT
  this.minHeap == orig(this.minHeap)
  this.maxHeap == orig(this.maxHeap)
  this.maximumSize == orig(this.maximumSize)
  this.queue[this.size-1] == this.queue[orig(this.size)]
  this.size >= 1
  this.modCount >= 1
  return == true
  this.queue[this.size-1] != null
  this.maximumSize > orig(this.size)
  this.maximumSize > orig(this.modCount)
  this.maximumSize > orig(size(this.queue[]))
  orig(element) in this.queue[]
  this.size - orig(this.size) - 1 == 0
  this.size != orig(this.modCount)
  this.modCount > orig(this.size)
  this.modCount - orig(this.modCount) - 1 == 0
  MMPQ.EVEN_POWERS_OF_TWO > orig(this.size)
  MMPQ.EVEN_POWERS_OF_TWO > orig(this.modCount)
  MMPQ.ODD_POWERS_OF_TWO < orig(this.size)
  MMPQ.ODD_POWERS_OF_TWO < orig(this.modCount)
  orig(this.size) <= size(this.queue[])-1
  orig(this.modCount) != size(this.queue[])
  size(this.queue[]) >= orig(size(this.queue[]))
  size(this.queue[])-1 != orig(size(this.queue[]))
  size(this.queue[])-1 >= orig(size(this.queue[]))-1
```

Listing B.5. Daikon output for MMPQ (5 of 10).

```
MMPQ.peek():::ENTER
  this.queue[this.size] == this.queue[this.modCount]
  this.queue[this.size] == this.queue[MMPQ.DEFAULT_CAPACITY−1]
  this.maximumSize == 2147483647
  size(this.queue[]) one of { 11, 12 }
  this.queue[this.size] == null
  this.size < MMPQ.DEFAULT_CAPACITY
  this.size < size(this.queue[])−1
  this.modCount < MMPQ.DEFAULT_CAPACITY
  this.modCount < size(this.queue[])−1
  MMPQ.DEFAULT_CAPACITY <= size(this.queue[])
  MMPQ.DEFAULT_CAPACITY >= size(this.queue[])−1

MMPQ.peek():::EXIT
  this.minHeap == orig(this.minHeap)
  this.maxHeap == orig(this.maxHeap)
  this.maximumSize == orig(this.maximumSize)
  this.queue == orig(this.queue)
  this.queue[] == orig(this.queue[])
  this.size == orig(this.size)
  this.modCount == orig(this.modCount)
  this.queue[this.size] == this.queue[this.modCount]
  this.queue[this.size] == this.queue[MMPQ.DEFAULT_CAPACITY−1]
  this.queue[this.size] == this.queue[orig(this.modCount)]
  this.queue[this.size] == orig(this.queue[post(this.modCount)])
  this.queue[this.size] == orig(this.queue[post(MMPQ.DEFAULT_CAPACITY)−1])
  this.queue[this.size] == orig(this.queue[this.modCount])
  this.maximumSize == 2147483647
  size(this.queue[]) one of { 11, 12 }
  this.queue[this.size] == null
  return in this.queue[]
  this.size < MMPQ.DEFAULT_CAPACITY
  this.size < size(this.queue[])−1
  this.modCount < MMPQ.DEFAULT_CAPACITY
  this.modCount < size(this.queue[])−1
  MMPQ.DEFAULT_CAPACITY <= size(this.queue[])
  MMPQ.DEFAULT_CAPACITY >= size(this.queue[])−1

MMPQ.peekLast():::ENTER
  this.queue[this.size] == this.queue[this.modCount]
  this.queue[this.size] == this.queue[MMPQ.DEFAULT_CAPACITY−1]
  this.maximumSize == 2147483647
  size(this.queue[]) one of { 11, 12 }
  this.queue[this.size] == null
  this.size < MMPQ.DEFAULT_CAPACITY
  this.size < size(this.queue[])−1
  this.modCount < MMPQ.DEFAULT_CAPACITY
  this.modCount < size(this.queue[])−1
  MMPQ.DEFAULT_CAPACITY <= size(this.queue[])
  MMPQ.DEFAULT_CAPACITY >= size(this.queue[])−1
```

Listing B.6. Daikon output for MMPQ (6 of 10).

```
MMPQ.peekLast():::EXIT
  this.minHeap == orig(this.minHeap)
  this.maxHeap == orig(this.maxHeap)
  this.maximumSize == orig(this.maximumSize)
  this.queue == orig(this.queue)
  this.queue[] == orig(this.queue[])
  this.size == orig(this.size)
  this.modCount == orig(this.modCount)
  this.queue[this.size] == this.queue[this.modCount]
  this.queue[this.size] == this.queue[MMPQ.DEFAULT_CAPACITY−1]
  this.queue[this.size] == this.queue[orig(this.modCount)]
  this.queue[this.size] == orig(this.queue[post(this.modCount)])
  this.queue[this.size] == orig(this.queue[post(MMPQ.DEFAULT_CAPACITY)−1])
  this.queue[this.size] == orig(this.queue[this.modCount])
  this.maximumSize == 2147483647
  size(this.queue[]) one of { 11, 12 }
  this.queue[this.size] == null
  return in this.queue[]
  this.size < MMPQ.DEFAULT_CAPACITY
  this.size < size(this.queue[])−1
  this.modCount < MMPQ.DEFAULT_CAPACITY
  this.modCount < size(this.queue[])−1
  MMPQ.DEFAULT_CAPACITY <= size(this.queue[])
  MMPQ.DEFAULT_CAPACITY >= size(this.queue[])−1

MMPQ.poll():::ENTER
  this.maximumSize == 2147483647
  size(this.queue[]) one of { 11, 24, 102 }
  MMPQ.DEFAULT_CAPACITY <= size(this.queue[])
  MMPQ.DEFAULT_CAPACITY != size(this.queue[])−1

MMPQ.poll():::EXIT
  this.minHeap == orig(this.minHeap)
  this.maxHeap == orig(this.maxHeap)
  this.maximumSize == orig(this.maximumSize)
  this.queue == orig(this.queue)
  size(this.queue[]) == orig(size(this.queue[]))
  this.maximumSize == 2147483647
  size(this.queue[]) one of { 11, 24, 102 }
  this.queue[this.size] == null
  this.maximumSize > orig(this.size)
  this.maximumSize > orig(this.modCount)
  orig(this.queue[post(MMPQ.DEFAULT_CAPACITY)−1]) in this.queue[]
  this.size < this.modCount
  this.size <= orig(this.size)
  this.size < orig(this.modCount)
  this.size <= size(this.queue[])−1
  this.modCount > orig(this.size)
  this.modCount >= orig(this.modCount)
```

Listing B.7. Daikon output for MMPQ (7 of 10).

```
  MMPQ.EVEN_POWERS_OF_TWO > orig(this.size)
  MMPQ.EVEN_POWERS_OF_TWO > orig(this.modCount)
  MMPQ.ODD_POWERS_OF_TWO < orig(this.size)
  MMPQ.ODD_POWERS_OF_TWO < orig(this.modCount)
  MMPQ.DEFAULT_CAPACITY <= size(this.queue[])
  MMPQ.DEFAULT_CAPACITY != size(this.queue[])−1
  return in orig(this.queue[])
  orig(this.size) <= size(this.queue[])

MMPQ.pollFirst():::ENTER
  this.maximumSize == 2147483647
  this.size >= 1
  size(this.queue[]) one of { 11, 24, 102 }
  this.queue[this.size−1] != null
  MMPQ.DEFAULT_CAPACITY <= size(this.queue[])
  MMPQ.DEFAULT_CAPACITY != size(this.queue[])−1

MMPQ.pollFirst():::EXIT
  this.minHeap == orig(this.minHeap)
  this.maxHeap == orig(this.maxHeap)
  this.maximumSize == orig(this.maximumSize)
  this.queue == orig(this.queue)
  size(this.queue[]) == orig(size(this.queue[]))
  this.queue[this.size] == this.queue[orig(this.size)−1]
  orig(this.queue[post(this.size)]) == orig(this.queue[this.size−1])
  this.maximumSize == 2147483647
  return != null
  size(this.queue[]) one of { 11, 24, 102 }
  this.queue[this.size] == null
  orig(this.queue[post(this.size)]) != null
  this.maximumSize > orig(this.size)
  this.maximumSize > orig(this.modCount)
  orig(this.queue[post(MMPQ.DEFAULT_CAPACITY)−1]) in this.queue[]
  this.size < this.modCount
  this.size − orig(this.size) + 1 == 0
  this.size < orig(this.modCount)
  this.size <= size(this.queue[])−1
  this.modCount > orig(this.size)
  this.modCount − orig(this.modCount) − 1 == 0
```

Listing B.8. Daikon output for MMPQ (8 of 10).

```
  MMPQ.EVEN_POWERS_OF_TWO > orig(this.size)
  MMPQ.EVEN_POWERS_OF_TWO > orig(this.modCount)
  MMPQ.ODD_POWERS_OF_TWO < orig(this.size)
  MMPQ.ODD_POWERS_OF_TWO < orig(this.modCount)
  MMPQ.DEFAULT_CAPACITY <= size(this.queue[])
  MMPQ.DEFAULT_CAPACITY != size(this.queue[])-1
  return in orig(this.queue[])
  orig(this.size) <= size(this.queue[])

MMPQ.pollLast():::ENTER
  this.maximumSize == 2147483647
  size(this.queue[]) one of { 11, 24 }
  MMPQ.DEFAULT_CAPACITY <= size(this.queue[])
  MMPQ.DEFAULT_CAPACITY != size(this.queue[])-1

MMPQ.pollLast():::EXIT
  this.minHeap == orig(this.minHeap)
  this.maxHeap == orig(this.maxHeap)
  this.maximumSize == orig(this.maximumSize)
  this.queue == orig(this.queue)
  size(this.queue[]) == orig(size(this.queue[]))
  this.maximumSize == 2147483647
  size(this.queue[]) one of { 11, 24 }
  this.queue[this.size] == null
  this.maximumSize > orig(this.size)
  this.maximumSize > orig(this.modCount)
  orig(this.queue[post(MMPQ.DEFAULT_CAPACITY)-1]) in this.queue[]
  this.size < this.modCount
  this.size <= orig(this.size)
  this.size < orig(this.modCount)
  this.size <= size(this.queue[])-1
  this.modCount > orig(this.size)
  this.modCount >= orig(this.modCount)
```

Listing B.9. Daikon output for MMPQ (9 of 10).

```
  MMPQ.EVEN_POWERS_OF_TWO > orig(this.size)
  MMPQ.EVEN_POWERS_OF_TWO > orig(this.modCount)
  MMPQ.ODD_POWERS_OF_TWO < orig(this.size)
  MMPQ.ODD_POWERS_OF_TWO < orig(this.modCount)
  MMPQ.DEFAULT_CAPACITY <= size(this.queue[])
  MMPQ.DEFAULT_CAPACITY != size(this.queue[])−1
  return in orig(this.queue[])
  orig(this.size) <= size(this.queue[])

MMPQ.size():::EXIT
  this.minHeap == orig(this.minHeap)
  this.maxHeap == orig(this.maxHeap)
  this.maximumSize == orig(this.maximumSize)
  this.queue == orig(this.queue)
  this.queue[] == orig(this.queue[])
  this.size == return
  this.modCount == orig(this.modCount)
  return == orig(this.size)
  return >= 0
  this.maximumSize > return
  this.modCount >= return
  MMPQ.EVEN_POWERS_OF_TWO > return
  MMPQ.ODD_POWERS_OF_TWO < return
  return <= size(this.queue[])
```

Listing B.10. Daikon output for MMPQ (10 of 10).

```
MMPQ.toArray():::ENTER
  this.maximumSize == 2147483647
  size(this.queue[]) one of { 11, 102 }
  this.queue[this.size] == null
  this.size < size(this.queue[])−1
  this.modCount != MMPQ.DEFAULT_CAPACITY
  MMPQ.DEFAULT_CAPACITY <= size(this.queue[])
  MMPQ.DEFAULT_CAPACITY != size(this.queue[])−1
  7 ∗ this.size + 7 ∗ this.modCount − 10 ∗ size(this.queue[]) − 2 == 0
  7 ∗ this.size + 7 ∗ this.modCount − 10 ∗ size(this.queue[])−1 − 12 == 0

MMPQ.toArray():::EXIT
  this.minHeap == orig(this.minHeap)
  this.maxHeap == orig(this.maxHeap)
  this.maximumSize == orig(this.maximumSize)
  this.queue == orig(this.queue)
  this.queue[] == orig(this.queue[])
  this.size == orig(this.size)
  this.size == size(return[])
  this.modCount == orig(this.modCount)
  this.maximumSize == 2147483647
  return != null
  return[] elements != null
  size(this.queue[]) one of { 11, 102 }
  this.queue[this.size] == null
  this.size < size(this.queue[])−1
  this.modCount != MMPQ.DEFAULT_CAPACITY
  MMPQ.ODD_POWERS_OF_TWO < size(return[])−1
  MMPQ.DEFAULT_CAPACITY <= size(this.queue[])
  MMPQ.DEFAULT_CAPACITY != size(this.queue[])−1
  7 ∗ this.size + 7 ∗ this.modCount − 10 ∗ size(this.queue[]) − 2 == 0
  7 ∗ this.size + 7 ∗ this.modCount − 10 ∗ size(this.queue[])−1 − 12 == 0
  7 ∗ this.modCount − 10 ∗ size(this.queue[]) + 7 ∗ size(return[])−1 + 5 == 0
  7 ∗ this.modCount − 10 ∗ size(this.queue[])−1 + 7 ∗ size(return[])−1 − 5 == 0
```

REFERENCES CITED

AMMONS, G., BODÍK, R., AND LARUS, J. R. 2002. Mining specifications. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*. ACM Press, New York, NY, USA, 4–16.

BOSHERNITSAN, M., DOONG, R., AND SAVOIA, A. 2006. From Daikon to Agitator: lessons and challenges in building a commercial tool for developer testing. In *ISSTA '06: Proceedings of the 2006 International Symposium on Software Testing and Analysis*. ACM Press, 169–180.

COK, D. R. AND KINIRY, J. R. 2004. ESC/Java2: Uniting ESC/Java and JML - progress and issues in building and using ESC/Java2. In *Construction and Analysis of Safe, Secure and Interoperable Smart Devices: International Workshop, CASSIS 2004*. Springer–Verlag.

COUSOT, P. AND COUSOT, R. 2004. *Basic Concepts of Abstract Interpretation*. Kluwer Academic Publishers, 359–366.

CSALLNER, C. AND SMARAGDAKIS, Y. 2006. Dynamically discovering likely interface invariants. In *Proc. 28th ACM/IEEE International Conference on Software Engineering (ICSE), Emerging Results Track*. ACM, 861–864.

CSALLNER, C., TILLMANN, N., AND SMARAGDAKIS, Y. 2008. DySy: Dynamic symbolic execution for invariant inference. In *Proc. 30th ACM/IEEE International Conference on Software Engineering (ICSE)*. ACM, 281–290.

DAHM, M. 2001. Byte code engineering with the BCEL API. Tech. rep.

ERNST, M. D. 2010. *Daikon Invariant Detector User Manual*. MIT Computer Science and Artificial Intelligence Laboratory. http://groups.csail.mit.edu/pag/daikon/download/doc/daikon.html.

ERNST, M. D., PERKINS, J. H., GUO, P. J., MCCAMANT, S., PACHECO, C., TSCHANTZ, M. S., AND XIAO, C. 2007. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming 69,* 1–3, 35–45.

FINDLER, R. B. AND FELLEISEN, M. 2001. Contract soundness for object-oriented languages. In *In OOPSLA '01 Conference Proceedings, Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 1–15.

GOOGLE. 2011. Guava: Google core libraries for Java 1.5+.

GUTTAG, J. V., HORNING, J. J., GARL, W. J., JONES, K. D., MODET, A., AND WING, J. M. 1993. Larch: Languages and tools for formal specification. In *Texts and Monographs in Computer Science*. Springer-Verlag.

GUTTAG, J. V., HORNING, J. J., AND WING, J. M. 1985. The Larch family of specification languages. *Software, IEEE 2,* 5, 24–36.

HANGAL, S. AND LAM, M. S. 2002. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering*. ICSE '02. ACM, New York, NY, USA, 291–301.

HOARE, C. A. R. 1969. An axiomatic basis for computer programming. *COMMUNICATIONS OF THE ACM 12,* 10, 576–580.

JACKSON, D. 2002. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol. 11,* 2, 256–290.

KICZALES, G., HILSDALE, E., HUGUNIN, J., KERSTEN, M., PALM, J., AND GRISWOLD, W. 2001. Getting started with AspectJ. *Commun. ACM 44*, 59–65.

KING, J. C. 1976. Symbolic execution and program testing. *Commun. ACM 19,* 7, 385–394.

KUZMINA, N. AND GAMBOA, R. 2007. Extending dynamic constraint detection with polymorphic analysis. In *Proceedings of the 5th International Workshop on Dynamic Analysis*. WODA '07. IEEE Computer Society, Washington, DC, USA, 1–.

LAMPORT, L. 1989. A simple approach to specifying concurrent systems. *Commun. ACM 32,* 1, 32–45.

LEAVENS, G. T. 1996. An overview of Larch/C++: Behavioral specifications for C++ modules. Tech. rep., DEPARTMENT OF COMPUTER SCIENCE, IOWA STATE UNIVERSITY.

LEAVENS, G. T. 1999. Larch/C++ reference manual. Version 5.41. Available in ftp://ftp.cs.iastate.edu/pub/larchc++/lcpp.ps.gz or on the World Wide Web at the URL http://www.cs.iastate.edu/˜leavens/larchc++.html.

LEAVENS, G. T. 2006. JML's Rich, Inherited Specifications for Behavioral Subtypes. In *ICFEM*, Z. Liu and J. He, Eds. Lecture Notes in Computer Science Series, vol. 4260. Springer, 2–34.

LEAVENS, G. T., BAKER, A. L., AND RUBY, C. 1998. Preliminary design of JML: A behavioral interface specification language for Java. Tech. rep.

LEAVENS, G. T. AND CHEON, Y. 2006. Design by contract with JML.

LEAVENS, G. T. AND NAUMANN, D. A. 2006. Behavioral subtyping, specification inheritance, and modular reasoning. Tech. Rep. 06-20b, Department of Computer Science, Iowa State University, Ames, Iowa, 50011. Sept.

LISKOV, B. H. AND WING, J. M. 1994. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems 16*, 1811–1841.

LORENZOLI, D., MARIANI, L., AND PEZZÈ, M. 2008. Automatic generation of software behavioral models. In *Proceedings of the 30th international conference on Software engineering*. ICSE '08. ACM, New York, NY, USA, 501–510.

MEYER, B. 1992. Applying "Design by Contract". *IEEE Computer 25,* 10, 40–51.

NIMMER, J. W. 2002. Automatic generation and checking of program specifications. Tech. Rep. MIT-LCS-TR-852, MIT Lab for Computer Science, 200 Technology Square. June.

NIMMER, J. W. AND ERNST, M. D. 2002. Automatic generation of program specifications. In *ISSTA 2002, Proceedings of the 2002 International Symposium on Software Testing and Analysis*. Rome, Italy, 232–242.

PERKINS, J. H. AND ERNST, M. D. 2004. Efficient incremental algorithms for dynamic detection of likely invariants. In *Proceedings of the ACM SIGSOFT 12th Symposium on the Foundations of Software Engineering (FSE 2004*. 23–32.

RICE, H. G. 1953. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society 74,* 2, 358–366.

SCHMIDT, D. A. 1998. Trace-based abstract interpretation of operational semantics. *Lisp and Symbolic Computation 10,* 3, 237–271.

SPINCZYK, O., LOHMANN, D., AND URBAN, M. 2005. Advances in AOP with AspectC++. In *SoMeT*, H. Fujita and M. Mejri, Eds. Frontiers in Artificial Intelligence and Applications Series, vol. 129. IOS Press, 33–53.

SPIVEY, J. M. 1989. *The Z notation: a reference manual.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

VOLKMANN, R. M. 2008. ANTLR 3. *Java News Brief*. http://jnb.ociweb.com/jnb/jnbJun2008.html.