SpecTackle: Inferring Partial Specifications Through Constraint-Based Dynamic Analysis

A Thesis

Presented in Partial Fulfillment of the Requirements for the Degree Master of Science in the Graduate School of The Ohio State University

By

Sean A Wedig, B.S.

Graduate Program in Computer Science and Engineering

The Ohio State University

2012

Master’s Examination Committee:

Neelam Soundarajan, Advisor
Bruce W. Weide
© Copyright by
Sean A Wedig
2012
Abstract

Formal specifications are widely recognized as beneficial, and in some niches, formal methods have been gaining ground. Unfortunately, many legacy systems have no formal specification – this is the “legacy code problem”. Many such systems could benefit from additional specifications – such specifications can serve as useful documentation, acting as educational tools for new users and aides to maintaining system integrity as the system is modified over its lifecycle. Even though training for and acceptance of formal methods is increasing, the sheer amount of unspecified legacy code is daunting. Clearly any assistance, automated or semi-automated, would be of considerable value.

Along a different route, automated testing and related techniques such as test-driven development are now recognized as a good practice in industry, and many systems have robust test suites for exercising a majority of their code base. Dynamic analysis systems such as Daikon and DIDUCE can take advantage of such test suites to infer specifications for systems under analysis, but they show sensitivity to the values of test inputs, possibly inferring incorrect invariants based on those values.

One possible mitigation approach is to allow hand-written specifications, thereby overriding incorrect or extraneous inference, but the above systems do not support such a technique. The work reported in this thesis is based on the idea that, by starting with a body of pre-specified core methods (rather than relying on statistical
confidence metrics as with systems such as Daikon), an automated system may be able to arrive at better specifications. How can we build upon such a foundation?

Interestingly, similar questions have been raised in type research. Rubydust approaches the issue of inferring static types for the dynamically typed Ruby language. Utilizing observations recorded during test runs, it employs a constraint-based dynamic analysis approach that builds upon given specifications of core Ruby libraries to infer static type specifications for classes built upon that foundation. Could a similar dynamic, constraint-based technique based on suitable information recorded during test runs be applied to other classes of specifications, in particular, behavioral specs?

It is this question that motivated the development of SpecTackle, a technique for automatically inferring partial specifications through constraint-based dynamic analysis (i.e., a technique for helping programmers tackle writing specs). Although the approach handles only a small subset of Java, the results reported in this thesis show that it promises to be more effective than statistical techniques such as Daikon because it does not have a similar sensitivity to test-input values.
To my wife, who went crazy keeping the rest of our lives running while I did this thing. I’ll get all the credit for doing the work, but she sacrificed nearly as much of her own time to make this happen.

And to my daughter, who gave us both a reason to do it all.
Acknowledgments

I would like to extend my sincere thanks to the providers of Soot and CVC3; without those tools, much of our implementation would have been vastly more difficult, if not insurmountably so. Additionally, I would especially like to thank Eric Bodden and the rest of the Soot community for their excellent documentation and tutorials, as well as their general openness and willingness to answer questions. Such support was invaluable as I began working with the complex framework.

I am deeply grateful to my advisor, Neelam Soundarajan, for his continued support for my work (and patience with my continued delays!). His academic experience has been absolutely essential during this process, and his calm, careful insistence for nothing short of excellence dramatically shaped the form this thesis has taken.

And lastly, a thanks to anyone else that supported this thesis, either directly or indirectly: Ted Pavlic, for the fantastic LaTeX template; Kelly Davis-Orr, for allowing flexibility in my work schedule; and any others that showed enough understanding to go easy on me in the rest of my life and/or provided review and proofreading. I doubt I could have completed this work without such a support network.
Vita

2004 ................................. B.S. Computer Science,
Case Western Reserve University

2004 - 2006 ......................... Applications Analyst,
Bridgestone / Firestone North American Tire, LLC

2006 - 2011 .......................... Web Application Developer,
The Ohio State University
Fisher College of Business

2011 - present ....................... Lead Application Developer,
The Ohio State University
Human Resources Technology

Fields of Study

Major Field: Computer Science and Engineering
# Table of Contents

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
</tr>
<tr>
<td>Dedication</td>
</tr>
<tr>
<td>Acknowledgments</td>
</tr>
<tr>
<td>Vita</td>
</tr>
<tr>
<td>List of Figures</td>
</tr>
<tr>
<td>1. Introduction</td>
</tr>
<tr>
<td>1.1 Motivations</td>
</tr>
<tr>
<td>1.2 Possible Solutions and Past Work</td>
</tr>
<tr>
<td>1.2.1 Parallels in Type Research</td>
</tr>
<tr>
<td>1.2.2 Generating Specifications from Test Runs</td>
</tr>
<tr>
<td>1.3 Organization</td>
</tr>
<tr>
<td>2. Conceptual Overview</td>
</tr>
<tr>
<td>2.1 Overview</td>
</tr>
<tr>
<td>2.1.1 More on Predicates and Predicate Variables</td>
</tr>
<tr>
<td>2.1.2 Constraints</td>
</tr>
<tr>
<td>2.1.3 Candidate Predicate Inference</td>
</tr>
<tr>
<td>3. Approach Details</td>
</tr>
<tr>
<td>3.1 A Single Expression Call</td>
</tr>
<tr>
<td>3.2 Straight-line Code</td>
</tr>
<tr>
<td>3.3 Layered Unannotated Methods</td>
</tr>
<tr>
<td>3.4 Branching Code, Loops, and Path Sensitivity</td>
</tr>
<tr>
<td>3.4.1 Loops</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>A small example program to demonstrate <em>Rubydust</em>’s capabilities.</td>
<td>5</td>
</tr>
<tr>
<td>3.1</td>
<td>A summary of the SpecTackle algorithm for observing expressions.</td>
<td>19</td>
</tr>
<tr>
<td>3.2</td>
<td>Straight-line code in a single method using only annotated expressions.</td>
<td>23</td>
</tr>
<tr>
<td>3.3</td>
<td>Straight-line code with multiple unannotated methods.</td>
<td>28</td>
</tr>
<tr>
<td>3.4</td>
<td>A simple example of branching Java code.</td>
<td>32</td>
</tr>
</tbody>
</table>
Chapter 1: Introduction

1.1 Motivations

The importance of formal specification and verification is widely recognized, especially for critical software and for hardware systems. However, while it appears that acceptance is spreading for use in the hardware domain, use of formal methods by software developers outside of a few specific niches is still limited. At best, a subset of methods or classes in any given system may be specified, but the majority of the system is not.

A major problem is that most software designers and developers find formal specifications difficult to understand and even more difficult to write. Experiments with modifying undergraduate education to include formal methods demonstrates positive results with making the techniques more approachable [9], but only time will tell if this leads to a wider-scale adoption of formal methods. Unfortunately, the current population of programmers generally has much more limited exposure, and so many systems go without the benefits of formal methods.

Worse, the improvement in education doesn’t directly address the “legacy code problem”. There are enormous amounts of software around the world that have been in use for years, even decades, and the vast majority of it is likely not specified,
or at least, not specified in a manner that lends itself to formal verification. Many of these systems cannot be wholesale recreated easily or inexpensively, and so it is unlikely that they will be recreated from the ground up to include such specifications. In order to reap the benefits of formal methods, precise specifications need to be developed for legacy code that can be of value for developers charged with maintaining or building upon the legacy systems (assuming, of course, that they’re comfortable reading the formal specs!).

1.2 Possible Solutions and Past Work

One possible approach is to simply train software engineers in formal methods and set them to the task of retrofitting specifications onto the legacy applications. While this approach could certainly work, the millions and millions (perhaps billions!) of lines of unspecified code present an increasingly enormous challenge for any would-be retro-specifiers. This raises one of the fundamental questions motivating this work: could we build some (semi-)automated means of generating the specifications for part or all of the legacy systems, thus easing that burden?

Meanwhile, automated testing has gained popularity in recent years, and it is becoming increasingly common to find suites of automated tests designed to exercise as much of a system’s code as possible. Testing has gained a strong foothold, and it is perceived by many as a “best practice” in software engineering. Thus our fundamental question may be revised as follows: Given the prevalence and positive perception of tests, could we take advantage of the existing test suites to infer specifications?
Daikon [5] and DIDUCE [8] both have attempted to infer system properties based on test run observations. While Daikon in particular provided inference for likely invariants that are similar to the specifications we wish to infer, Daikon’s approach is susceptible to reporting invariants that are more indicative of “accidental” assumptions in the test suite. Such assumptions may not reflect actual assumptions in the system under test, instead capturing invariants in the values of the test suite itself. In other words, Daikon may present invariants that are invalid or not useful.

This issue is compounded by the fact that Daikon doesn’t provide any means of overriding the inferred specifications in order to build a collection of “approved” specs. As with many automated systems, the inference may yield extraneous information or the information may be phrased in awkward ways which are useful to the inference engine, but which are not easily read or used by a human programmer. It would be desirable to allow a skilled hand to modify or override the specifications inferred, so that extraneous information can be cleared away and specs can be rephrased in a way that is most useful. How can we provide a skilled programmer a means of cleaning up our automated results?

A part of the answer, at least, lies at the core of our approach: start with a foundation of accepted specifications, and then use behavioral observations to build out the collection of annotated (i.e., specified) methods. Pre-written specifications can then either be manually created by a programmer or taken from an automated system (with appropriate review and cleaning, if needed), allowing more flexibility in how specifications are defined and added to the system.

Making use of a body of pre-written specifications especially comes into play when we consider the common scenario of developers building systems upon a
foundation of libraries, frameworks, or other “core” methods and classes. Broadly speaking, such core methods are often well-documented – perhaps even formally specified! If a method \( m \) only calls pre-annotated, formally-specified methods, could we combine the specifications of the called methods with an understanding of language constructs such as if/then/else statements to build a conglomerate specification for \( m \)? By extension, if all methods in a system eventually call into pre-specified framework methods, i.e. only build upon a foundation of specification, could we recursively build specifications for all methods in the system? That is the central question we wished to answer through this research.

1.2.1 Parallels in Type Research

A similar problem exists for dynamically typed languages such as Ruby when thinking about parameter and return types. Though the language is dynamically typed, programmers often program as though these types are actually static. Is it possible to observe the usage of types in a given method so that we can infer what type the programmer was intending as the implicit static type? Can we note how a variable is used (as a method call receiver, as an argument, etc.) in order to constrain the types a variable may have? Interestingly, many of the core classes of Ruby (e.g, Array, Numeric, String) typically used extensively by Ruby systems have methods and operators with parameter and return types that are statically defined [1]. Can we utilize the static definitions of the core in order extend the set of type-specified methods?

This is precisely the question Jong-hoon An et al. investigated with their tool Rubydust [2], specifically addressing the problem of inferring static types for method
parameters and return values in Ruby. They introduced constraint-based dynamic analysis, an approach which uses sub-typing constraints generated through observation of a test run to limit the possible types for method signatures. After the test run, the observed constraints are passed to a solver which attempts to satisfy the constraints with inferred types. Failure to satisfy all the constraints indicates that there is an inconsistent type usage in the system – a potential bug.

To demonstrate Rubydust's approach, consider a small example (inspired by Figure 2 from [2]). Consider the code in Figure 1.1, where we have a class A with method foo which will be exercised by executing the test suite (in this case, a single statement) at line 9. The class B's implementation is not relevant, but assume that B.new returns a new object, b.

```ruby
1  class A
2     # foo : α_x → α_.foo_ret
3     def foo(x) # x = (b : α_x)
4         r = 23.5 + x # r = (23.5 + b : Numeric)
5         return r # Numeric ≤ α_.foo_ret
6     end
7  end
8
9  A.new.foo(B.new) # B ≤ α_x
10     # ret = (23.5 + b : α_foo_ret)
```

Figure 1.1: A small example program to demonstrate Rubydust's capabilities.

When we first execute the program, starting at line 9, Rubydust generates a subtype constraint, type(b) ≤ α_x, to capture the observation that the type of b must be a subtype of the parameter x's static type. Rubydust introduces a type variable for the unknown type, denoted α_x. Type variables are similarly introduced for unknown return types, e.g., α_foo_ret for foo's return type; foo's full type is listed.
on line 2. Inference proceeds by constraining the type variables based on how associated variables are used during execution.

As we enter \texttt{foo}, the argument values are wrapped with proxy objects that associate type metadata with the value; in this case the argument \texttt{b} is associated with \(\alpha_x\). Wrapping lets \textit{Rubydust} track how various types are used by associating types (or type variables) with program variables. These observations lead to constraints on the type variables.

For example, on line 4, we perform an addition and generate a constraint \texttt{Numeric} \(\leq \alpha_{\text{foo,ret}}\). In Ruby, addition is treated as a call to \texttt{Numeric}’s + method, which is annotated to require a single \texttt{Numeric} argument. As a result, we can build a constraint that requires that \(\alpha_x\) is a subtype of \texttt{Numeric}, thus constraining part of \texttt{foo}’s specification based on the “foundational” specification of \texttt{Numeric}. After the addition is executed, the value is associated with the specified return type of + (i.e., \texttt{Numeric}) and then that value is returned at line 5. \textit{Rubydust} generates another constraint to capture that the return type allows \texttt{Numerics} based on the associated type of the value returned during this test run. Finally, another round of wrapping occurs, and so as execution returns from \texttt{foo}, the return value is wrapped with \texttt{foo}’s return type, \(\alpha_{\text{foo,ret}}\).

In order to solve the constraints, \textit{Rubydust} organizes them into a directed graph, with edges from child types to parent types. The tool then attempts to find a least solution for the constraints. For arguments, it selects an upper bound of transitive successors, and for return values it selects a lower bound. For our example, this lets us infer that \((\alpha_{\text{foo,ret}} = \texttt{Numeric})\) and that \((\alpha_x = \texttt{Numeric})\) as well.
1.2.2 Generating Specifications from Test Runs

Our approach to building specifications from test runs was inspired, in part, by the *Rubydust* approach. We start with a collection of specifications for methods and operators, and then, based on observations of suitable test runs, generate constraints from which we can infer specifications. As we will demonstrate, our approach does allow us to generate suitable specifications for systems under test, and a prototype implementation of the approach, named *SpecTackle*, seems to gather and solve constraints efficiently. These early results are promising, but many questions about the approach and how to generalize it to more complex data structures and operations are still to be answered. For now, we’ve limited ourselves to Java methods using only immutable, arithmetic types without abstraction, with a small body of pre-specified operators and library methods. In order to take advantage of the test suites available, we build constraints dynamically through observation of test runs, instead of performing a static analysis.

1.3 Organization

We first present an overview of the logical underpinnings of *SpecTackle* in Chapter 2. To show the sequence of observations and metadata gathering, we demonstrate the technique in a series of examples in Chapter 3. The architecture of a partial implementation of *SpecTackle* is detailed in Chapter 4, while Chapter 5 discusses *SpecTackle* in relation to similar approaches and other related work. Finally, in Chapter 6 we discuss some of the system’s limitations and future work with the technique.
Chapter 2: Conceptual Overview

2.1 Overview

Our approach to inferring specifications starts with the assumption that we have a set of annotated methods, $A$, the annotations being trusted, pre-written specifications. We have another set, $U$, of unannotated methods for which we wish to infer parameter preconditions and return-value postconditions. For each method $u \in U$, we introduce undefined predicates to represent the per-parameter preconditions and return-value postconditions. Through observation of the program under test, we then generate constraints on these predicates, and, based on these constraints, we build candidates for the pre- and postconditions of $U$.

An unknown predicate, called a predicate symbol, denotes a relationship over its parameters; the constraints built are thus designed to limit the possible relationships. To build constraints on the predicate symbols, we associate metadata with every variable used in the program during test runs. The metadata captures current assertions about the variable (in the form of a predicate) and the path taken through the method during the current execution. For a variable $v$, predicates come in two forms: as a fully defined predicate, taken from the specification of an annotated method $a \in A$, or as a predicate symbol. The former occurs when $v$ is
passed to or returned from the annotated \(a\), while the latter occurs when \(v\) is passed to or returned from an unannotated method \(u \in U\). Information on the path taken through a method body is captured as a sequence of program blocks visited, and conditions that were met at each program branch point.

As a variable \(v\) crosses a method threshold (i.e., is used as an argument to or returned from a method) we generate constraints on \(v\)’s predicate, \(\text{pred}(v)\), in order to limit the space of possible relationships. Specifically, when \(v\) is used as the \(i\)-th argument to a method \(m\), we build a constraint to state that the current predicate associated with \(v\) must imply the precondition of \(m\)’s \(i\)-th parameter. In other words,

\[
\text{pred}(v) \Rightarrow \Phi_{m,i} \tag{2.1}
\]

where \(\Phi_{m,i}\) is the precondition mentioned above. Note that \(m\) may be either be annotated or not. If \(m \in A\), then \(\Phi_{m,i}\) represents the specified precondition. If \(m \in U\), \(\Phi_{m,i}\) is a predicate symbol representing the unknown relationship.

Since this implication was observed during a test run, we assume it to be true, and so (2.1) can be treated as a constraint on the predicate \(\Phi_{m,i}\) that must be satisfied. When a value \(v'\) is returned, we similarly constrain the return-value postcondition of \(m\), \(\Psi_m\), with

\[
(v' = \text{ret}_m) \Rightarrow \Psi_m \tag{2.2}
\]

where \(\text{ret}_m\) is the symbol for \(m\)’s return value. Constraints are further modified based on the path conditions observed during execution to fully capture the method’s behavior. See Section 2.1.2 for more detail.

As may have been apparent in the above text, preconditions and postconditions in our context differ slightly from their traditional usage – preconditions are broken
into *per-parameter* preconditions and postconditions are specifically related to the return value of the method. The former design evolved out of the practical desire to associate predicates, including precondition predicates, with specific variables. The latter is for similar reasons, but in any case, it is justified by the assumption that we are working with non-mutable types in methods with no side effects.

Beyond constraints, we also capture the influence of variables on one another as the program executes through what we call *breadcrumbs*. Breadcrumbs are simple predicates that are solely used for capturing assignment of values to variables. They are captured immediately after executing an assignment, and they allow us to reconstruct how operations on values affect the final values returned from methods. Breadcrumbs are primarily useful during the constraint-solving stage of inference so we may convert variables that are purely internal to methods into equivalent forms that only make use of symbols on the method thresholds (i.e., the input parameters and return values) or constants.

Once we have observed all the test runs of the program and collected the body of constraints, we then produce candidates for the predicate symbols that should satisfy all of the constraints gathered. Further, we also attempt to confirm the candidates as valid to show that all the constraints are satisfied. Assuming that we can show the candidates to satisfy all the constraints, we present them as our inferred pre- and postconditions. More detail on the candidate predicate creation and validation is in Section 2.1.3.
2.1.1 More on Predicates and Predicate Variables

Our approach is built on the association of formal logical predicates to concrete program variables using a thin, intuitive concrete-to-logical mapping (e.g., the `double` type in Java is mapped to $\mathbb{R}$, the `int` type is mapped to $\mathbb{Z}$, etc.). As stated before, a predicate captures assertions about its associated variable, and in most cases, this will capture the source of the given variable, such as a predicate “$(a = b \times c)$” for the variable $a$. However, predicates associated with a variable do not necessarily need to be stated in this “assignment-like” structure, and they may capture any logical assertion about the associated variable.

Predicates are drawn from two sources: annotated methods that have pre-written, trusted specifications so that the predicates are known before analysis, and undefined placeholder predicate symbols. It is the latter that we are primarily interested in constraining during our analysis, though constraints on known predicates can indicate flaws in specification or potential bugs that did not manifest during test runs.

Annotated methods’ specifications are trusted – they are assumed to be correct, even if they say something contradictory, and they are never checked by our system for correctness$^1$. When associated with a variable, they are associated in the form supplied.

Unannotated methods by definition don’t have predefined specifications, so in order to properly constrain the predicates of the specification, we introduce appropriate predicate symbols that we then use throughout our analysis. Let $u$ be

$^1$In principle, our system could be coupled with a formal verification system such as [14] so that specifications written by hand could be validated as correct, too.
a method in $U$, and let $n$ be the number of parameters of $u$. We introduce undefined predicates $\Phi_{u,i}$ for the precondition for the $i$-th parameter of $u$ and $\Psi_u$ for the return-value postcondition predicate of $u$. Note that for any given method, we have a number of precondition predicate symbols, but only one postcondition predicate symbol. All the same usage of predicate symbols applies - they are associated with variables during execution and constraints like (2.1) are generated using them.

### 2.1.2 Constraints

All constraints are composed of three pieces: the path conditions which led to the observance of the constraint ($conds$), the predicate being constrained, and the limiter which provides a constraint. However, how they are combined is slightly different, depending on when the constraint is being generated.

If the constraint is being generated at the time of calling another method (e.g., executing the statement “bar($y$, $z$);”), then we build constraints to capture that the current branch decisions, $conds$, led to the observation that the current predicate of each argument, $pred(arg_i)$, must satisfy the preconditions of each of the method $m$’s $n$ parameters, that is

$$conds \Rightarrow (pred(arg_i) \Rightarrow \Phi_{m,i}) \text{ for } (i = 1 \ldots n)$$

(2.3)

which reduces to

$$(conds \land pred(arg_i)) \Rightarrow \Phi_{m,i} \text{ for } (i = 1 \ldots n)$$

(2.4)

When we observe a value being returned, we capture a constraint of the form

$$(conds \Rightarrow (ret_m = v)) \Rightarrow \Psi_m$$

(2.5)
Intuitively, this captures that the paths taken during execution led to $v$ being returned as $m$’s return value (i.e., $ret_m = v$), and that this should imply the post-condition of $m$, $\Psi_m$.

During a method $m$’s execution, constraints of the form in equation (2.4) are generated for all statements in $m$’s body, regardless of whether the statements are calls to annotated methods. As such, constraints can be used not only to limit possible predicates for predicate variables, but also as a means of collecting some basic verification conditions that can be checked for validity in order to demonstrate bugs that may not have manifested during a test run. However, our primary focus is on generating specifications, so we do not flag such errors in our current implementation.

As will be shown in Chapter 3, constraints of the postcondition variety are only generated when $m$ is unannotated; since we fully trust the annotations of methods in $A$, we do not make observations during $m$’s execution. As such, we also only generate this kind of constraint when executing unannotated methods.

### 2.1.3 Candidate Predicate Inference

When attempting to infer predicates, we first pass the collection of generated constraints and breadcrumbs to a theorem prover to assert the observations made. Since we observed each of the constraints during the test run, and the test run was successful, we assume them to have been satisfied with the parameters we observed. This is based on the fact that, were the observations not satisfied, then execution would have encountered some sort of error. Since we do not allow for exceptions, it would likely be an error that halted execution, which means that we
would not have reached the solving step at all. Based on that assertion, we can then start building and checking candidates for the predicates.

It should be noted that in the below discussions of the specific candidate building algorithms, we exclude the usage of predicate parameter symbols for clarity reasons. In all cases, our logical predicates include symbols representative of the context in which they were generated, capturing the logical equivalent of the arguments or parameters, return value symbols, etc.

**Building Candidates for Precondition Predicate Variables**

For a given method \( m \in U \) with a single parameter \( x \), we introduce an unknown precondition predicate symbol \( \Phi_{m,0} \). This predicate symbol will be involved in constraints in two forms:

\[
(\text{conds}_1 \land \Phi_{m,0}) \Rightarrow \Phi_{\text{other}_1,i}
\]  

(2.6)

and

\[
(\text{conds}_2 \land \Psi_{\text{other}_2}) \Rightarrow \Phi_{m,0}
\]  

(2.7)

where \( \Phi_{\text{other}_1,i} \) is the precondition predicate of a method \( \text{other}_1 \)'s \( i \)-th parameter, and \( \Psi_{\text{other}_2} \) is the post condition predicate of a method \( \text{other}_2 \).

When building a candidate for \( \Phi_{m,0} \), we gather all the constraints of similar form to equation 2.6. For each constraint, we then extract the conjunction of path condition assertions (\( \text{conds}_1 \) above) and the consequent for the constraint (\( \Phi_{\text{other}_1,i} \)). Using the extractions as a tuple (e.g., \( (\text{conds}_1, \Phi_{\text{other}_1,i}) \)), we build a clause for a candidate as

\[
\text{conds}_1 \Rightarrow \Phi_{\text{other}_1,i}
\]
The full candidate is then built by taking the clauses extracted from \( n \) constraints and conjoining them to give a candidate \( \phi_{m,0} \) as

\[
\begin{align*}
conds_0 & \Rightarrow \Phi_{\text{other}_0,i_0} \\
conds_1 & \Rightarrow \Phi_{\text{other}_1,i_1} \\
& \quad \vdots \\
conds_n & \Rightarrow \Phi_{\text{other}_n,i_n}
\end{align*}
\]

(2.8)

The reasoning behind such a structure for \( \phi_{m,0} \) is that we are interested in building an aggregation of how parameters are used in order to build a complete constraint on the possible allowed values for arguments - in other words, to build the precondition of the parameter.

While \( \phi_{m,0} \) as stated will obviously satisfy the constraints of the form in equation 2.6, we also will need to ensure that the rest of the constraints of the form in equation 2.7 are satisfied by this selection. If there are any constraints in the latter form that are invalid with \( \phi_{m,0} \) as the candidate, then that likely indicates that there is an error in the specification.

As an intuitive “proof”, consider a constraint \( C \) such that \( C := [(conds \wedge \Psi_{\text{bar}}) \Rightarrow \Phi_{\text{foo}}] \). Let \( \phi_{\text{foo}} \) be a candidate we’ve constructed, and assume that \( C \) is invalid if \( \phi_{\text{foo}} = \Phi_{\text{foo}} \). \( C \) being invalid means that under the assertions of \( conds \), the possible return values from \( \text{bar} \) (which satisfy \( \Psi_{\text{bar}} \)), do not necessarily satisfy \( \Phi_{\text{foo}} \). In other words, because of our thin abstract mapping, this means that return values from \( \text{bar} \) should cause an error when used as an argument to \( \text{foo} \). However, as stated above, this would result in halting execution, which leads to a contradiction - we cannot be solving for a predicate variable if execution halted during the test run.
Instead, such a state may also be caused by an unsound abstraction, such as an invalid specification.

**Building Candidates for Postcondition Predicate Variables**

We take a similar approach to building candidates for postconditions predicate symbols. For a given method \( m \in U \) with a single parameter \( x \), we introduce an unknown postcondition predicate symbol \( \Psi_m \). Let \( \Phi_{other,i} \) be the precondition predicate of a method \( other \)'s \( i \)-th parameter. The predicate symbol \( \Psi_m \) will be involved in constraints in two forms:

\[
(conds_1 \Rightarrow (ret_m = var)) \Rightarrow \Psi_m
\]

which is generated just before returning a variable \( var \) from \( m \) with a series of path condition decisions captures in \( conds_1 \), and

\[
(conds_2 \wedge \Psi_m) \Rightarrow \Phi_{other,i}
\]

which is generated when the value returned from \( m \) and used as the \( i \)-th argument to the method \( other \).

As above, we take a subset of all the constraints involving the predicate variable for which we’re trying to solve, and we extract information to build clauses of the candidate. For postcondition predicates, we are primarily interested in the values that are returned, and so we extract tuples \((conds_1, (ret_m = var))\) from the constraints that are similar in form to equation 2.9 and build clauses in the form

\[
(conds_1 \Rightarrow (ret_m = var))
\]

Similar to the precondition candidates, we build a candidate postcondition predicate, \( \psi_m \), from the clauses by joining them with a conjunction, as shown in equation
2.11:

\[(\text{conds}_0 \Rightarrow (\text{ret}_m = \text{var}_0)) \land (\text{conds}_1 \Rightarrow (\text{ret}_m = \text{var}_1)) \land \ldots \land (\text{conds}_n \Rightarrow (\text{ret}_m = \text{var}_n))\]  \hspace{1cm} (2.11)

A similar line of reasoning to that regarding \(\phi_m,0\) applies to \(\psi_m\) and its satisfaction of constraints similar to 2.9 and 2.10. Constraints formed as in 2.9 are clearly satisfied, while those similar to constraints formed like 2.10 should also be satisfied, unless there is an error in the specifications.
Chapter 3: Approach Details

This section contains more detail on how and when we associate predicates and build constraints. We start by discussing the observations we make around a single expression call, and how this is modified whether it is an annotated expression, unannotated expression, or an assignment statement (3.1). We then demonstrate how the technique handles successively more complicated examples: a simple straight-line code with fully annotated expressions (3.2), with multiple unannotated methods (3.3), and finally with branching code, including if/then constructs and loops (3.4).

3.1 A Single Expression Call

To demonstrate the timing of constraint generation, predicate association, and the creation of breadcrumbs, we start with a simple example. As stated, the majority of our observations focus on expressions and assignments of values in the system under test, and so as a result, they focus on the moments when we cross from caller to callee and back, in other words, when we cross method thresholds.

Consider an assignment statement of the form $x = m(a)$ for an expression $m$ with a single argument $a$ with a value of $v$. The expression (in this case, a method) $m$ has a single parameter $p$ and a body of $e$ which, when executed, yields
some returned object \( v' \). The primary operations taken for all such statements are *constraining*, *associating*, and *creating breadcrumbs*, and the order in which they are applied for the above-mentioned assignment is summarized in Figure 3.1.

Figure 3.1: A summary of the SpecTackle algorithm for observing expressions.

“Associate” steps (2 and 5) are where we assign predicates to variables within a given method’s body. “Constrain” steps (1 and 4) are those where we generate the constraints that limit the possible predicates for our unknowns. Step 6, Breadcrumb creation, only occurs when the statement is an assignment, but it does exactly as it states - it creates a breadcrumb to capture the flow of data. While there are
similarities within the categories of steps, the reasoning behind each instance of the step is subtly different and is detailed below.

Because our approach was inspired by the constraint-based approach taken by *Rubydust*, it bears some similarities. Our “associate” step is analogous to *Rubydust*’s “wrap” step, and the two types of “constrain” steps are related. In contrast, though, *Rubydust* has no concept (nor need) of breadcrumbs or similar observations, but the added complexity necessary for supporting behavioral constraints led to the introduction of step 6 for *SpecTackle*.

Before entering the body of *m*, we generate constraints in step 1. The first constraint, labeled (a) is as described earlier, where we constrain the possible predicates for the precondition predicate symbol of *m*’s first parameter, that is, $\Phi_{m,0}$. This is based on the intuitive understanding that by accepting argument $a$ for parameter $p$ (i.e., the first parameter), $a$’s current predicate must necessarily imply $\Phi_{m,0}$.

As the method threshold is crossed and we enter the callee (*m*), we associate predicate symbols with the parameter variables that receive the values used as arguments. Specifically, the local parameter $p$ is associated with the predicate symbol for $m$’s first parameter precondition, $\Phi_{m,0}$, within the context of the callee. Within the predicate we maintain that the parameter name, $p$, is the symbol used in the logical predicate, and in general, for straight-line code, the logical symbol used in predicates for variables will always match the variable name\(^2\). The association is essentially an observation that, for whatever the predicate $\Phi_{m,0}$ is determined to

\(^2\)As will be demonstrated when we discuss branching code (section 3.4), path-specific symbols are introduced when there are multiple possible logical statements using a particular variable, and so it will not be a 1-to-1 mapping from variable to logical symbol.
be, we can assert that $p$ satisfied it during this particular execution of $m$’s body. Later, as we use $p$ in the method, knowing that it is associated with the precondition predicate variable allows us to constrain that unknown predicate.

In step 3, the method body $e$ is executed, eventually resulting in a return value $v'$ which has some predicate $\text{pred}(v')$. For simplicity in this overview description, we will assume that $e$ is straight-line code, and so $\text{conds}$, a conjunction of the path conditions observed during this particular execution of $m$, is simply treated as $\text{TRUE}$. A more complete description of how such branching code is handled is in section 3.4.

In step 4, we add constraint (d.) in the context of the callee, i.e., in terms of a return value $\text{ret}_m$ and $m$’s parameters. Most importantly, we capture that $v'$ was the value returned. This constraint must also take into account the conditions that led to $v'$ being returned, and so we include an implication that the path conditions we observed imply the specific return value. Finally, that implication ($\text{conds} \Rightarrow (\text{ret}_m = v')$) should as a whole also imply the postcondition of $m$, $\Psi_m$, because this is one specific execution out of all possible paths through $m$.

As we return back to the caller’s context in step 5, we replace the associations of the return value to reflect the change in context to the caller method. By re-labeling return values in this way, we provide a layer of abstraction such that the details of the specific return value removed, so that from the perspective of our analysis, we only are concerned that the value was returned from $m$, and that it must meet the constraints of postcondition of $m$.

Finally, for assignment statements, we use step 6 to generate a breadcrumb capturing the flow of values. In this case, we note that $a$ was used as an argument
to \( m \) and execution yielded a value that was stored in \( x \). The breadcrumb will be most useful during the solving stage, for rephrasing candidates using constraints such as (d.) in terms of method parameters and return values. However, since breadcrumbs are only useful for values that continue to flow in the program, we only capture them on assignments\(^3\).

### 3.2 Straight-line Code

To demonstrate the algorithm with a full example, consider the code in Figure 3.2. The method \( \text{foo} \) performs a simple, self-contained calculation which depends only on operators (which are treated as annotated expressions), while the main method acts as the test suite used for exercising the method under test with some sample data. Since we are only dealing with straight-line code in this example, \( \text{conds} \) will always be set to \( \text{TRUE} \). For clarity, we exclude it from generated constraints.

Since the arguments and return values of main methods are generally part of bootstrapping, they are excluded from inference. As such, we begin our example just before entering \( \text{foo} \)'s body, but before switching context from the caller. We introduce logical symbols \( x \) and \( y \) for the parameters of \( \text{foo} \), and \( \text{ret}_{\text{foo}} \) for its return value. Since \( \text{foo} \) is unannotated, predicate symbols \( \Phi_{\text{foo},0} \), \( \Phi_{\text{foo},1} \), and \( \Psi_{\text{foo}} \) are also introduced. For the constant values passed to \( \text{foo} \) as arguments \( 5.0 \) and \( 10.0 \), we do not introduce symbolic variables, and instead the constants are used directly in predicates.

\(^3\)Given that we assume no side effects for methods, we find it unlikely that we would have any standalone expression statements like \( m(a) \);, but it’s still syntactically valid. Further, it actually could affect the inferred specifications for preconditions, so we allow such code, even if we don’t generate breadcrumbs on non-assignments.
class Example1 {
    public static double foo(double x, double y) {
        double local = x / y;  // Φ_foo,0(x, y) ⇒ TRUE
        // BC: local = x / y
        double midStep = local;  // no constraints generated
        // BC: midStep = local
        local = midStep + 1.0;  // (midStep = x / y) ⇒ TRUE
        // BC: local₁ = midStep + 1.0
        return local;  // (ret_foo = local₁) ⇒ Ψ_foo(ret_foo, x, y)
    }
    public static void main(String[] args) {
        double z = foo(5.0, 10.0);  // TRUE ⇒ Φ_foo(5.0, 10.0)
        // BC: z = foo(5.0, 10.0)
    }
}

Before entering foo, we then generate constraints as described earlier in step 1, and listed on lines 38 and 39. As a convention, we assume that for every constant \( c \), \( \text{pred}(c) := \text{TRUE} \). As a result, the 5.0 and 10.0 values used as arguments simply add \( \text{TRUE} \) as their predicates to the constraints generated. Since we assume the
logical constraints are satisfied, constraints on lines 38 - 39 constraints reduce to $\Phi_{foo,0}(5.0, 10.0)$ and $\Phi_{foo,1}(5.0, 10.0)$. Intuitively, this captures the constraint that foo’s preconditions must be satisfied with the arguments (5.0, 10.0).

Once we’ve completed step 1, we then cross the method threshold into foo’s body. As we enter foo, we associate new symbolic variables and predicates with the double-type variables to reflect the new context. As described in step 2, the arguments to x and y are associated with $\Phi_{foo,0}(x, y)$ and $\Phi_{foo,1}(x, y)$, respectively. This provides a layer of abstraction for our analysis, as we do not consider the specific values used as arguments once we cross the method threshold; instead, all later constraints generated from using x and y will only consider the variables’ predicate associations.

In step 3, we proceed to execute the body of foo, first executing the division operation on line 18, which, as mentioned above, is treated as an annotated expression (see lines 1 - 9 for the specs of operators used in this example). The annotation provides preconditions for both of the parameters as well as the postcondition. Using the annotation and the predicates for x and y, we generate constraints on the preconditions, observing that $\Phi_{foo,0}(x, y) \Rightarrow TRUE$ and $\Phi_{foo,1}(x, y) \Rightarrow (y \neq 0)$. After execution of the expression, we associate the predicate from the annotation with local, and then, since it is an assignment, we also generate a breadcrumb.

On line 23 we execute another assignment statement, but this time with an expression that is simply another variable with no actual operation. Such re-assignments are treated as though they were an annotated expression with no

\[4\]It’s fair to note that the Java specification [6] doesn’t actually require the denominator to be non-zero, but we believe that NaN, \texttt{POSITIVE_INFINITY}, and \texttt{NEGATIVE_INFINITY} are rarely desirable results. For this reason (and for the sake of demonstration), and we defined our specification to with that in mind.
parameters. In essence, it is a method that simply returns a value equal to local’s value with a “postcondition” predicate equal to \( \text{pred}(\text{local}) \). As a result, we copy \( \text{pred}(\text{local}) \) and associate the clone with \( \text{midStep} \). Then, since this is an assignment statement, we also generate a breadcrumb to capture the relationship between \( \text{midStep} \) and \( \text{local} \).

Executing another annotated expression on line 27, we generate two more constraints based on the number of parameters to +. Addition has no actual precondition constraints on allowed values for either parameter, so our constraints simply require that the predicates on the arguments imply \( \text{TRUE} \). There are two items worthy of note here: the reuse of the \( \text{local} \) variable and the creation of the constraint using \( \text{pred}(\text{midStep}) \).

The reuse of the concrete \( \text{local} \) actually results in a mapping to a unique logical symbol \( \text{local}_1 \). For our mapping of concrete code to logic to be sound, we need to make sure that we do not create inconsistent logical equalities when we assert our breadcrumb observations. As such, when we capture observations about line 27, we introduce a new unique symbol, \( \text{local}_1 \), which is separate from \( \text{local} \) (and, hypothetically, if we had more lines of code that assigned to \( \text{local} \), we would introduce more symbols \( \text{local}_2, \text{local}_3 \), etc. as needed).

A short aside about the symbol name: This symbol name is generated through the use of \textit{Shimple}, a single-static assignment (SSA) variant of \textit{Jimple}. \textit{Jimple} is an intermediate representation of Java bytecode provided as part of Soot [15]; more details on our usage of Soot are discussed with the implementation details in Chapter 4, but briefly, it is a representation for Java programs that is at a level of abstraction between the bytecode and the source code. \textit{Shimple} is almost identical
to Jimple except for its SSA nature. As such, this naming is, in-part, rooted in our implementation, but any other implementation of SpecTackle would need to employ a similar technique.

The logical symbol $local_1$ is used for all logical constructs created at this point that would reference the mapped symbol for $local$, including the predicate associated with $local$ and the breadcrumb generated. Since later constraints generated from the usage of $local$ will use its predicate, those constraints will also reference $local_1$. In this way, our approach respects the flow of operations in a given method, or, as coined by the Rubydust developers, it is flow-sensitive.

The latter item of note is that we make use of the current predicate associated with a variable when we generate line 27’s constraints. Thus, using $pred(midStep)$, we yield the (perhaps not helpful) constraint $pred(midStep) \Rightarrow TRUE$, or equivalently, $(midStep = x/y) \Rightarrow TRUE$. However, even if the “constraint” is always satisfied, it demonstrates the use of an intermediate variable in an expression, and how a variable’s currently associated predicate – regardless if it came from an annotated or unannotated expression – is used to build the constraint. This is the primary avenue by which specifications from core methods flow out and influence the inferred predicate symbols, and indeed, how predicate symbols even influence each other through constraints. Specifications become associated with variables, and as those variables are used as arguments or return values for unannotated methods, the constraints we generate link the parameter symbols of the unannotated methods with the specifications.

On line 32, this is exactly what we see happening: we return $local$ (mapped to $local_1$), and we generate a constraint on $\Psi_{foo}$ as described in step 4 above. The spec
from 27 bubbles up because of its association with \( \text{local}1 \), and then it is involved in a constraint as \( \text{local}1 \) is returned. While this example will always result in the same return value following the same path through \( \text{foo} \), later examples utilizing branching code will make this point more self-evident.

As we return from \( \text{foo} \), we cross the method threshold back into the caller’s context. For step 5, we now associate \( \Psi_{\text{foo}} \) with the value returned from \( \text{foo} \), abstracting away the specific value returned. Note however, that \( \Psi_{\text{foo}} \) has been re-contextualized in terms of \( \text{main} \)’s context, so that the predicate symbol’s parameters are in terms of the arguments passed to \( \text{foo} \) and the variable receiving the return value. The breadcrumb also creates a logical mapping of \( \text{foo} \) that, for the purposes of solving, is left as an undefined function. This allows us to refer to the abstraction of \( \text{foo} \) within our inferred predicates.

Once execution is complete, we hand the collection of generated constraints and breadcrumbs over to our solving engine, which produces candidates for the unknown predicate symbols, validates them, and displays the results. More about the candidate inference process is discussed in Section 2.1.3. For this example, we are able to infer that

\[
\Phi_{\text{foo},0}(x, y) := TRUE
\]

\[
\Phi_{\text{foo},1}(x, y) := (y \neq 0)
\]

for the preconditions of \( \text{foo} \), and for the postcondition, we infer

\[
\Psi_{\text{foo}} := (\text{ret}_{\text{foo}} = \text{local}_1)
\]

(3.1)
However, when presenting the results of inference for foo’s specification, we hide the constraints and breadcrumbs generated as purely internal to foo. So instead of presenting Equation 3.1 in that manner, we make use of the breadcrumbs (prefixed with “BC:” in the comments of Figure 3.2) to perform appropriate substitutions until 3.1 is phrased in terms of the method parameters and return value:

$$\Psi_{foo} = (\text{ret}_{foo} = ((x/y) + 1)) \quad (3.2)$$

### 3.3 Layered Unannotated Methods

The example of Section 3.2 demonstrates the simplest of examples – one where there is no branching to the code and all expressions executed by the unannotated method are in $A$. While useful for demonstration purposes, it is obviously somewhat contrived. Instead, consider another example where we have more than one layer of methods in $U$ calling one another, as shown in Figure 3.3.

```java
class Example2 {
    public static double foo(double x, double y) {
        double m = bar(x, y);
        y = 23.0;
        return baz(x, y);
    }

    public static double bar(x,y) {
        return x / y;
    }

    @PredicateSymbols({"$ret", "$a", "$b"})
    @ReturnValuePredicate("$ret > ($a * 15)")
    public static void baz(@Precondition("NOT($a = 5)") a,
                            @Precondition b) {
        // ... body of baz (not shown) ...
    }
}
```

Figure 3.3: Straight-line code with multiple unannotated methods.
The approach for Figure 3.3 is much the same - an entry point of foo leads to generating constraints as other expressions are executed. At the call to bar, though, we generate constraints that include two predicate symbols: \((\Phi_{foo,0} \Rightarrow \Phi_{bar,0})\) and \((\Phi_{foo,1} \Rightarrow \Phi_{bar,1})\). Fortunately, this doesn’t pose a problem for the algorithm; we simply allow such a constraint and then ensure it is satisfied during solving. However, this generally means that there is an enforced order for which parameter symbol we solve first. In this case, the dependency is clear-cut, and so the spec for bar must be inferred before foo. In particular, execution of bar proceeds as described earlier, and the constraints generated lead to a natural inference so that bar’s spec matches that of the division operator, which then plays a part in foo’s spec.

Another piece of note is that even though we generate constraints, generate a breadcrumb, and associate a predicate with the variable \(m\), the variable is never actually used. The constraints themselves are emitted and captured in a separate data structure; the presence or absence of a receiving variable does not affect whether we emit constraints (though it does affect if we emit breadcrumbs). In fact, line 3, rewritten as the equivalent code “bar(x, y);” would generate the same constraints.

Line 5 returns the return value from the annotated baz. Because we convert Java programs into Shimple, line 5 is converted into an assignment to a generated Shimple local variable (named something like $d0), which is then returned. In fact, because Jimple and its variants (such as Shimple) are based on three-address code (TAC), all complex, multi-step expressions are separated into their intermediate steps, so even expressions like “\((a*b) + ((x/y) - 5)\)” are broken down into single-operator expressions whose results are assigned to Shimple-generated local
variables. Since our analysis looks purely at the *Shimple* code, we can’t differentiate between TAC assignments and Java code that was originally written in such a manner. As a result, *SpecTackle* generates a breadcrumb to capture the implicit assignment to the temporary variable just before it is returned.

Execution of baz before returning demonstrates a programmer-specified, (i.e., annotated) method. Since we are analyzing Java programs, it was a natural fit to use annotations as the means of providing specs. In this specific example, baz is specified as follows:

\[
\begin{align*}
\Phi_{baz,0} & := (a \neq 5) \\
\Phi_{baz,1} & := \text{TRUE} \\
\Psi_{baz} & := (ret > (a \times 15))
\end{align*}
\]

where *ret* is the logical mapping for the return value of baz.

The *@Precondition* and *@ReturnValuePredicate* annotations are straightforward. However, the *@PredicateSymbols* annotation is slightly less obvious. It is intended to provide an ordered list of symbols used by all the pre- and postcondition annotations. The order is assumed to be consistently “(Return-value symbol, Parameter symbols)” . The symbol list is an aid to *SpecTackle* for interpreting the raw text added by the user, so that it can properly pick out the variable symbols which may change, depending on the context of the predicates.

Constraints generated at the return of *foo* on line 5 will utilize the specification for baz as described earlier, while the generation of constraints will halt during baz’s body (because baz ∈ A). The decision to halt generation of constraints is actually done at instrumentation-time – the transformations performed on the bytecode are
only performed on unannotated methods. So baz is left as-is, leaving us to gather no constraints from its *immediate* body. Note, however, that if baz were to call an unannotated callee method (such as foo), we would generate constraints while inside that unannotated callee (but not from within baz).

Returning from baz generates return-value constraints (making use of the specification), and we proceed to terminate the test run. Other than the previously-mentioned order of inference for predicate symbols, the solving algorithm doesn’t change when multiple unannotated methods are involved.

### 3.4 Branching Code, Loops, and Path Sensitivity

The previous examples are all simple straight-line methods in which there is no branching code. However, naturally, we need to handle such code in order to begin approaching real systems.

To do this, *SpecTackle* captures some additional information from the basic constraints and breadcrumbs described earlier. As described in the overview (Chapter 2), we also capture a conjunction of all path conditions satisfied (aka *conds*) and a sequence of code blocks visited during the execution. Combined with the SSA names of *Shimple*, this allows us to create unique, path-sensitive logical names, which in turn allows inference over all observed executions.

Consider the code in Figure 3.4. The method foo is broken into a series of control graph code blocks based on how execution may proceed. Block 0 is succeeded by Blocks 1 and 2 (depending on the path condition evaluation), which are both succeeded by Block 3.
Since we can only build candidates for predicate symbols based on what’s been observed, we must cover all paths through `foo` during our test runs in order to infer a sound specification. While it’s planned as future work to produce a formal logical analysis of the approach (which supposedly would include some attempt at a proof of soundness), it’s likely that any specifications inferred would only be sound for the paths observed, with the corollary that if all paths are observed during test runs, then the specs inferred are sound. Even without a formal treatment, it’s intuitively clear that the code coverage of the test suite will play a key component in the correctness of inferred specs. For `foo`, then, we must have a test suite that covers both branches of the if/then/else of lines 7 - 15.

For the first execution of `foo`, assume that `(x ≠ 0)` so that we enter Block 1 from Block 0. Observation by `SpecTackle` proceeds exactly the same as with the straight-line code examples, with two key differences: 1.) upon crossing the branch point,
SpecTackle observes the logical assertion of the path condition (in this case, \((x \neq 0)\)) and adds it to \(\text{conds}\), and 2.) observes the ID of the block we’re entering, and adds it to the block sequence. Crossing from Block 0 to Block 1 sets the block sequence to simply \(<1>\), since it’s assumed that every execution will start in Block 0.

As described in Chapter 2, \(\text{conds}\) is a key component of all constraints generated. By collecting the path conditions observed, it also captures the logical assertions under which constraints are satisfied\(^5\). At line 9, we’ll generate the expected constraint involving \(\Phi_{\text{foo},0}(x)\) and \((x \neq 0)\), with the addition of the observed path condition to yield a constraint

\[
[(x \neq 0) \land \Phi_{\text{foo},0}(x)] \Rightarrow (x \neq 0)
\]

which is always satisfied. In essence, because of the placement of the path condition to guard against erroneous inputs, we don’t actually constrain \(\Phi_{\text{foo},0}\).

We will, however, constrain \(\Psi_{\text{foo}}\) with the condition chain, despite the protective condition. For a test run where \((x \neq 0)\), we’ll generate the full constraint

\[
[(x \neq 0) \Rightarrow (\text{ret}_{\text{foo}} = \text{temp}_3)] \Rightarrow \Psi_{\text{foo}}(\text{ret}, x)
\]

where \(\text{temp}_3\) is another unique name introduced via SSA. For execution down the \texttt{else} path, we’ll generate

\[
[(x = 0) \Rightarrow (\text{ret}_{\text{foo}} = \text{temp}_3)] \Rightarrow \Psi_{\text{foo}}(\text{ret}, x)
\]

. However, earlier in the execution, each of those block sequences would have generated different breadcrumbs which would allow us to show that \([\langle 10/x \rangle = \text{temp}_3 = -x]\). This clearly wouldn’t be sound!

\(^5\)For Block 0, \(\text{conds} := \text{TRUE}\). If there is no branching, \(\text{conds}\) will never change, and thus non-branching code’s constraints will simplify the \(\text{conds}\) term out.
To address the issue that arises from variable re-use over several executions, there are two levels of uniqueness used for symbols names: SSA variable names ensure that re-assignments in straight-line code logically unique, while a unique symbol based on the block sequence ensure uniqueness over multiple executions on divergent paths. This combination creates what we call block sequence-unique symbols. For example, for the first run where \((x \neq 0)\), the logical symbols introduced for \texttt{temp} are \texttt{temp}_{1,\leq 1} and \texttt{temp}_{3,\leq 1,3}, at lines 9 and 18, respectively. \texttt{temp}_{1,\leq 1} indicates that it is the first “extra” \texttt{temp} symbol introduced via the SSA of \texttt{Shimple}, and that it was observed for the block sequence \(<0,1,\rangle\). Similarly, \texttt{temp}_{3,\leq 1,3} is the third \texttt{temp} symbol (\texttt{temp}_{2} is on line 14) with a block sequence of \(<0,1,3,\rangle\). These are distinct from each other as well as from the symbols \texttt{temp}_{2,\leq 2} and \texttt{temp}_{3,\leq 2,3} which would be introduced for a test run where \((x = 0)\).

As a result, over the two test runs exercising both branches of \texttt{foo}, we actually generate the more unique (and useful!) constraints

\[
[(x \neq 0) \Rightarrow (ret_{\texttt{foo}} = temp_{3,\leq 1,3})] \Rightarrow \Psi_{\texttt{foo}}(ret, x)
\]

and

\[
[(x \neq 0) \Rightarrow (ret_{\texttt{foo}} = temp_{3,\leq 2,3})] \Rightarrow \Psi_{\texttt{foo}}(ret, x)
\]

Combined with the breadcrumbs to restate and simplify the constraints in terms of the method threshold variables, they become

\[
[(x \neq 0) \Rightarrow (ret_{\texttt{foo}} = 10/x)] \Rightarrow \Psi_{\texttt{foo}}(ret, x)
\]

and

\[
[(x \neq 0) \Rightarrow (ret_{\texttt{foo}} = -x)] \Rightarrow \Psi_{\texttt{foo}}(ret, x)
\]

which allows for inference of an accurate return value specification for \texttt{foo}.
3.4.1 Loops

Loops for SpecTackle are treated in the same manner as any other branching code. Because of the block sequence-unique symbols, we can collect logically unique constraints and breadcrumbs over multiple loop iterations. This is essentially equivalent to “unrolling” the loops and observing execution of the straight code. On the positive side, this means that we can cheaply and efficiently perform inference over loops – the dynamic approach means that we completely sidestep the issue of inference over infinitely many possible paths of a loop. Of course, the downside is quite significant, in that in this early version of the algorithm our inferred specs will only be accurate for the precise number of iterations observed. Arguably, such specs written “as-is” are not useful to a developer wishing to generate specifications for unannotated methods.

Instead, the output of specs for loops is better served as a tool to help programmers see patterns in specifications. A developer can add or remove test cases to see how inferred specifications are modified, and then it’s possible that patterns will start to emerge. Since we allow custom specification, the programmer can then go in and enter their own specs, based on either the outputs from SpecTackle or from more traditional methods of arriving at specifications for loops.

The current approach sacrifices accuracy for efficiency and simplicity, and the inferred specs only provide some basic assistance to developers attempting to develop robust, generalized specs for methods with loops. As such, loops are one of the biggest horizons of future work for this research; some possible directions this might take are discussed in Section 6.2.
Chapter 4: Implementation Details

In this chapter, we discuss a prototype implementation of SpecTackle. Our system is composed of three main components: 1.) a bytecode transformation engine built upon the Java optimization framework Soot [15], 2.) a mechanism for collecting runtime observations called the constraint engine, and 3.) a constraint solving engine built as a wrapper around the CVC3 SMT solver [3].

We start with the design goals for the system (4.1), and then detail the expected workflow for using SpecTackle on some existing Java code (4.2). We then discuss the overall architecture (4.3), transformations and instrumentation of bytecode (4.4), and then conclude with some details about the solver engine’s translation from SpecTackle to CVC3 (4.5).

This actually discusses the second attempt at implementing the SpecTackle approach. Before our Soot-based implementation, we attempted an aspect-oriented programming approach using the popular AspectJ framework. We ultimately abandoned the approach, as we encountered difficulty in achieving the fine-grained control necessary to gather the observations we wanted to perform, especially when concerning ourselves with branching code. Because of the low-level capabilities provided by Soot, as well as its excellent documentation and community, we decided to implement SpecTackle with this framework instead.
4.1 Design Goals

Beyond implementing a proof-of-concept for the algorithm, we had some underlying goals for the system. Not all of the goals are fully realized, as the system is still in the early stages of development, but these ideals have been guiding many internal decisions for the implementation.

1. **Easy to apply to existing code.** As long as the code under test doesn’t hit the limitations of *SpecTackle*, there should be few roadblocks beyond installation.

2. **Low overhead relative to normal execution.** The instrumentation for dynamic observation and execution of the solving algorithm shouldn’t add excessive overhead to the test runs. Ideally, this means that the observation and solving would scale about linearly with the size of the system under test.

3. **Human-readable output.** Results from inference, error messages, and warnings should all be developed with a human developer in mind. The fewer esoteric, hard-to-read messages the better.

4.2 The Workflow: Using SpecTackle

The expected workflow for using *SpecTackle* is as follows:

1. Prepare the files for input: write Java source code for the system under test and (optionally) compile it.

2. Use *SpecTackle* to instrument the bytecode with *Soot*-based transformations via the *SpecTackleSootMain* class

3. Run the transformed class files with a standard JVM, and observe the results.
The steps are detailed below.

### 4.2.1 Preparation of Input Files

Compilation is optional, as *Soot* can process `.java` source files directly. In fact, passing raw source code has some advantages, as *Soot* can extract extra information such as line number tags by examining `.java` files directly.

However, noting that source files are not always available, compiled class files are also acceptable input. In particular, *SpecTackle* works best when the `.java` source files are compiled with debugging symbols turned on. While not strictly necessary, including the debugging symbols allows *Soot* to produce predicates and constraints utilizing the original symbols names, which greatly helps in readability. This is, however, optional as well.

### 4.2.2 Instrumenting Source Files

The *SpecTackle* instrumentation is performed by running a custom `main` method that wraps calls to *Soot* with 4 custom body transformers. To instrument, pass the input files (as compiled bytecode `.class` and/or raw source `.java`) to the *SpecTackle* main class, `SpecTackleSootMain`. For our analyses we used the following options:

- `cp {source directory};{bytecode directory}` to provide *Soot* the path of the classpath for the system under analysis that should be used during transformation.

- `exclude spectackle` excludes the *SpecTackle* packages from analysis. Not strictly necessary, but it drastically speeds up transformation by excluding the moderately complex libraries.

38
-no-bodies-for-excluded ensures that SpecTackle and the JDK aren’t loaded for transformation (when combined with the -exclude option above).

-p jb use-original-names:true helps with readability of the output by telling Soot to use the original variable names in source files instead of the generic names normally in bytecode.

-via-shimple is necessary for SpecTackle to function, as the custom transformations are only applied during the Shimple transformation phase.

-src-prec java tells Soot to prefer the usage of .java source code (instead of bytecode), so that we can gain the benefit of original variable names and other debugging information.

Additional options were also used for debugging purposes (e.g., -output-dir and -dump-body) or to simplify the analysis command-line (e.g. by simplifying the classpath with -prepend-classpath), but they do not have an effect on the analysis. SpecTackle has no custom command-line options itself; it simply provides custom transformations for Soot. In practice, the prototype transparently passes all command-line arguments to Soot. For more details on the options available see [10].

The input files will be transformed as detailed in Section 4.4, resulting in modified class files that can be executed as-is.

4.2.3 Executing the SpecTackle Analysis

Because the modified class files now include calls to the constraint and solver engines, the SpecTackle class files (including Soot and CVC3) need to be included
in the classpath for the execution. Otherwise, execution of the system under test post-transformation is exactly the same as pre-transformation.

Execution of the test suite should proceed as normal, but the instrumentation allows the data structures of the constraint engine to collect observations. Once the test suite completes, right before exiting from the main method, SpecTackle’s the solving engine takes over, translating the run-time constraints into a format usable by CVC3. The solving engine performs the necessary candidate predicate building and validation. As predicates are inferred, they are displayed via a simple STDOUT exports with simple prompts, while errors and warnings emitted during the solving step are written to STDERR.

4.3 Architecture

The Java packages that compose the prototype of SpecTackle are structured at a high level into three large collections: those for classes relating to transformations, those for gathering constraints, and those for solving the constraints. They are each discussed at a high level in the following sections.

4.3.1 Transformation Code Architecture

The transformation classes are all custom body transformers from Soot (meaning that they all extend soot.BodyTransformer). When included as part of a transformation phase, BodyTransformers are passed the bodies of methods under analysis in the form of one of Soot’s internal representations (in our case, Shimple), and, as the name implies, they perform various transformations on the methods.
There are four transformers used by SpecTackle with a common ancestor class (see Section 4.4 for details of the transformations). AroundExpressionTransformer performs the transformations around statements within a method’s body. The InternalMethodBoundaryTransformer transforms the entry and exit of methods to associate predicates and generate constraints, respectively. The Condition-ObservationTransformer implements the transformation that adds calls to methods to observe path conditions and the sequence of blocks visited. Finally, the SolveConstraintsMainMethodTransformer transforms the main method to ensure that SpecTackle kicks off the constraint solving step once the test suite is completed.

Each of the BodyTransformer classes in SpecTackle inherit from a parent transformer class, BaseSpecTacklerTransformer. This class collects common utility functions used by each of the transformers. Of particular interest are the methods that support tagging statements. In Soot, extra metadata can be added to most bytecode structures, including single statements, with objects referred to as Tags. For SpecTackle, we tag every inserted statement and observation with the SpecTacklerInstrumentationTag tag. This ensures that the four transformations, which are run serially, do not attempt transformations around inserted statements and observations; anytime a transformation encounters such a tagged statement, it simply skips it.

4.3.2 Constraint Engine Classes

Used primarily during the test runs for generation of constraints, breadcrumbs, and other observations, the ConstraintEngine class is the “front-line” interaction with the system under test. Most interactions call some static method of
ConstraintEngine, passing it base types, Objects, Strings, and PathSequence objects (see below). This was motivated by a desire that the signatures of the ConstraintEngine’s methods would be as simple as possible, thus reducing the amount and complexity of the modifications made to the bytecode.

Since a sequence of blocks visited is really dependent on the stack frame, we actually instrument methods under analysis with local instances of PathSequence objects. These objects collect a list of program block IDs as well as the conditions needed to visit the blocks – effectively all the information necessary to understand the sequence of paths taken for a given method’s execution. Again, with the interest of limiting the complexity of instrumentation, these objects are created from, and passed to ConstraintEngine’s methods, where nearly all of the manipulation takes place.

This architecture provided for much more agility as we work on the prototype – changes to straight Java code in the ConstraintEngine are far simpler to perform than modifications to Soot transformations. Additionally, by creating an local variable to contain such stack-frame-specific observations such as the path sequence, we let the JVM do all the work of relating the observations to a given stack frame, saving a great deal of effort in accurately modeling the execution stack.

The ConstraintEngine is supported by a group of data structures, mostly singletons (e.g. for collecting constraints via the EmittedConstraints class). Currently, this includes a DataPredicates singleton data structure that contains the associations of predicates to the variables within a given method. Note that this is within a method, and not within a given stack frame. This immediately causes problems with recursive methods, and it is for this reason that the implementation
doesn’t support them. This was an early addition to the prototype, and it will be updated to utilize a stack-frame sensitive approach like PathSequence’s approach described above.

Additionally, ConstraintEngine works closely with a cadre of helper classes that handle pockets of functionality outside the its main purpose of executing the constraint gathering logic and providing a simple interface to systems under analysis. Helpers include those for translating Soot-type signatures into Java’s code-reflection objects (such as Method and Class), retrieving and parsing specifications from annotated methods, and providing “faux methods” for operators so that we can provide specifications for the operators that are functionally equivalent to other methods.

### 4.3.3 Solver Engine Classes

The SolverEngine class does not interact directly with the ConstraintEngine as may be expected, but instead retrieves the gathered constraints from the singletons mentioned above. Its primary role is to process the gathered observations in the proper order while (with the support of helper classes) translating SpecTackle objects into CVC3 logical constructs. The SolverEngine is handles the overarching algorithm, but it is also supported by two “sub-engines” for handling candidate inference for pre- and postcondition predicate symbols. This sub-engine model provides an extensible framework under which SolverEngine can be extended in the future to support inference for other predicate symbols (such as loop invariants). Interaction with CVC3 is done through the Java API in order to interact directly
with the ValidityChecker engine, which allows for finer-grained control over the interactions.

4.4 Transformations and Instrumentation

Instrumentation of Java bytecode by SpecTackle is performed using custom method body transformers using the excellent Soot optimization framework. Because we wish to take advantage of the semi-unique names introduced with SSA, we chose the internal representation of Shimple over the standard Jimple for Soot. There are four custom transformations implemented for SpecTackle: the around expression transformer, the internal method boundary transformer, the condition observation transformer, and the main-method constraint solver transformer.

The most complex of the transformers is that of the around expression transformer. Around each expression calls are inserted into the bytecode that will, at run-time, collect observations about all the arguments passed to an expression. The names and types are in essence hard-coded into the inserted ConstraintEngine calls - essentially a “static observation” of the original code. However, the inserted calls also pass the argument values to the constraint engine, which is the true dynamic observation of interest. This transformer also inserts all post-expression calls, such as doing predicate assignment to the returned value and inserting calls to generate breadcrumbs, if needed. In essence, it inserts all the calls necessary for the caller-context constrain/wrap/create steps as laid out in Chapter 3. Most of the complexity in this class is because of the sheer number of expression types that must be supported. Even in our prototype, which only handles a small subset of
possible Java expressions, this requires a surprising amount of code, leaving it the largest of the three transformers by far.

Transformations around expressions are complemented by the transformations for the internal method boundaries. These transformation perform the transformations to enable the callee-context associate and constrain steps. It inserts ConstraintEngine calls to assign predicates to the method’s parameters, and to generate postcondition constraints at every exit point of the method.

Path conditions and block sequence observations are enabled through the transformations for observing path conditions and block visits. The transformer for this performs a very simple analysis of the block sequences to add observations about the conditions under which each program block may be visited in the method. It also inserts dynamic observation calls at the head of each block to capture that program execution visited the block.

The last transformer, for inserting calls to solve the constraints gathered is the simplest of the lot. It only transforms the main method of the system under test, and its only goal is to find the exit points of the method (return statements or completion of the method). Just before the exit points, this transformer inserts a call to solve the constraints and output the inference results. We currently assume, then, that the test suite will only complete via the exit points mentioned above, and not through other means, such as System.exit().

4.5 SpecTackle to CVC3 Translation

During the observation phase, we gather several SpecTackle objects of interest that represent our observations, e.g. predicates, constraints, and breadcrumbs.
These each have their own Java objects, and, in order to work with CVC3, these all must be converted into appropriate CVC3 expressions.

We convert most variables using a thin mapping from a Java types to a logical symbol. The symbol name uses both the SSA variable name, as well as the path sequence (see 3.4) to maintain appropriate uniqueness. Additionally, some uniqueness related to Java scope is also observed, so methods with the same parameter names, for example, will have unique atomic terms in CVC3.

**Breadcrumbs** are translated into named terms in CVC3. This means that we can take advantage of the optimizations that CVC3 uses for quick substitution from named terms down to atoms and operations. In particular, because breadcrumbs are recordings of internal assignments for a method, the substitution replaces local variable symbols with symbols for the method parameters, which allows us to display inferred predicates in such term with very little work⁶.

Breadcrumbs that involve predicate symbols are converted into an assignment-like breadcrumb with a function placeholder. So for example, a breadcrumb that is from the postcondition of `foo`, Ψ_{foo}, will actually create an undefined function `foo` in CVC3 of an appropriately mapped function type, which can then be used in postconditions.

**Constraints** are just logical implications, and so we convert them into assertions in CVC3. In this incarnation, we do not validate if the assertions are, in fact, valid, though we assume them to be. This is a potential source of problems with inference if there are contradictory specifications. Future versions will address this.

⁶The downside is that this implementation only works with breadcrumbs (and by extension, annotated return value predicates) of the form “⟨receiver⟩ = ⟨expression⟩;”. This is really a limitation of the current implementation, though; the SpecTackle approach does not actually limit. Future work is planned to perform the substitution outside of CVC3.
Predicates in general are not converted by themselves, but instead are converted if they play a part in a constraints (in which case they are converted as part of the constraint). When they are converted, it may take place in a few different ways. Annotated get converted as-is. In fact, we assume annotations for SpecTackle are written in CVC3-readable format. (Ideally, this will be replaced with a more portable syntax in a future version.) Predicate symbols are created as undefined $\lambda$-expressions, which is then used directly in the constraint.

When a candidate predicate is created for a predicate symbol, we query CVC3 to confirm that a bijection between the candidate and the predicate symbol is valid. If so, we then generalize the equality by asserting the bijection, thereby assuming it holds for all values.
Chapter 5: Related Work

Static approaches to inference of system information has been pursued in a variety of systems [12, 13, 4]. Often, such approaches are built upon a foundation of generated program traces and extraction of information from the possible traces. However, scalability of such approaches quickly becomes a problem – statically generating all program paths that may lead to a method call-site produces an exponential number paths (the so-called “path-explosion” problem). Much has been done to address this problem (e.g., the approximations and probabilistic methods used in [11]).

Dealing with path-explosion means that many static approaches focus on carefully culling the number of paths analyzed, leading to a trade-off between precision and speed (or even tractability of a technique at all!). Many approaches use a mining-based approach, looking for commonalities in state abstractions near method call-sites to build confidence of the inclusion of a given predicate $p$ in a precondition for the method. Such techniques build confidence about invariants over a large number of possible paths; this provides robustness for handling erroneous program code – e.g., a given predicate $p$ might not be present at a call-site though it is actually valid precondition, but a statistical approach may still infer $p$
if it is common at most other call-sites. The downside is that such techniques must then rely on some confidence threshold to produce likely invariants.

On the other hand, dynamic approaches such as SpecTackle can avoid the path-explosion problem and completely avoid unrealizable paths. However, this comes at the cost of being reliant on the quality of test suites and other means of exercising systems under analysis. As touched on in Section 1.2, Daikon is a notable example of a dynamic approach. It bears some similarities with static approaches, as it also follows a mining-approach. However, instead of relying on collections of predicates extracted statically, Daikon gathers an execution trace including variable values. This collection is mined using a predefined set of potential invariants (e.g., this.x ≥ y) for which confidence metrics are created based on the execution trace.

Because of its reliance on actual values of program variables, Daikon differs from SpecTackle in the predicates it presents as likely invariants. In Daikon’s case, the invariants are similar to those produced from static techniques (by bearing a statistical confidence), but its dynamic nature means that which likely invariants are presented is affected by the quality of the test suite and the diversity of input values (i.e., false invariants are possible if x always happens to have the same relationship with y in the test suite).

By way of comparison, SpecTackle is also affected by the quality of the test suite, but instead of being sensitive to specific input values and their relationships, the important metric is that of path coverage. This limits the space of possible states for which tests must be written, and can easily provide guidance to programmers on how to improve a test suite for SpecTackle – ensure that the test suite contains
inputs to exercise every possible path of every method. While SpecTackle is path-sensitive, it is also modular in that it is only necessary to obtain path-coverage for the paths through each method (not the program as a whole), thus avoiding the path-explosion problem.

Xie and Notkin have integrated Daikon with other tools for generating unit test data to iteratively augment and improve the test suites it uses [16]. Perracotta is another system [17] that takes a similar approach as Daikon, but arrives at temporal specifications.

DIDUCE [8] also infers invariants dynamically, but unlike Daikon, it continually checks invariants as a program runs. When counterexamples to hypotheses (i.e., potential invariants) are found, the predicates are relaxed to allow for the newly observed behaviors. In addition to producing possible invariants, this also allows for interactive usage, where anomalies can be detected and examined, providing developers insights into what’s changed (e.g., just before a program crash when a precondition is violated).

Given these different approaches, a natural question that a practitioner might raise is, which of these approaches would be most effective for dealing with real systems? In our view, the answer to that question would be that specification-inference engines like SpecTackle (and those listed above) would be part of a larger suite of tools focusing on improving software quality in an arrangement analogous to tools that assist programmers in writing source code and the co-aligned tools for verifying syntax and compiling it. Verifiers such as RESOLVE are pushing the boundaries in this regard, but the “grand challenge” of a simple, push-button remains elusive [14]. In the meantime, SpecTackle can contribute to improving
software quality by assisting developers in creating precise specifications of new and, perhaps more importantly, legacy code.

A final consideration worth noting regarding the integration of such systems is that approaches such as *RESOLVE* operate with useful levels of abstraction for their specifications. Until now, we have ignored abstraction in our work, focusing on the feasibility of the approach, but the question of generating specifications that are at suitable higher-levels of abstraction is essential in dealing with large, complex systems. We will consider this briefly in Section 6.2.2.
Chapter 6: Conclusions

In this thesis, we presented SpecTackle, a proof-of-concept implementation of a constraint-based dynamic analysis algorithm for inferring behavioral specifications. We detailed the approach’s logical underpinnings and specifics about the implementation. Although much work remains to be done, (some of which is discussed below), our research has produced promising results. It has shown that SpecTackle has the potential of being an extremely useful tool to help software designers to develop precise specifications for large bodies of both new and legacy code.

In Section 6.1, we discuss some of SpecTackle’s key limitations and some possible approaches to addressing them, and in 6.2, we consider means of expanding the scope of programs SpecTackle can analyze. We are also working on making our prototype implementation of SpecTackle available to other researchers. Our hope is that there will be enough public interest in the tool in order to continue development of both the technique and the system.

6.1 Limitations

The work on the SpecTackle approach reported in this thesis is intended to be a proof of concept demonstrating the use constraint-based dynamic analysis for
behavioral specs. In order to allow for a manageable avenue of research, we intentionally limited the types of Java programs that can be analyzed to simple arithmetic using real numbers, integers, and boolean values with only base types (i.e., no mutable objects). This allowed us to focus on the core algorithm for gathering and solving for constraints and breadcrumbs, without worrying about the complexity of modeling object types. Such modeling brings with it at least a need to model the Java heap (or otherwise deal with aliasing issues), to address methods with side-effects, and to handle mutable object fields, all of which are complex, open problems in and of themselves. In such a strongly object-oriented language such as Java, this obviously limits the practical application of SpecTackle to a much smaller subset of Java methods; allowing arbitrary, mutable types is therefore one of the first targets for expansion of the algorithm.

As described in Chapter 3, inference over loops is done in the simplest manner possible by directly gathering constraints, effectively “unrolling” the loops during observation. Naturally, this can lead to unsound inference of specifications. Thus, as mentioned earlier, the current incarnation of SpecTackle should be considered (with respect to its prototypical state) as a tool that helps developers see patterns in specs, especially assertions in loops, over multiple executions of the same method. One possible approach would be to allow developers to provide, on the basis of observed patterns, additional information (such as loop invariants and progress metrics) in a manner similar to providing other annotations for SpecTackle. This additional information could then be incorporated into the approach so that the tool can further refine its inferred specifications. However, we also believe there is potential in using constraint-based approaches for inferring loop invariants without
such additional information, and we describe an overview of a possible avenue of investigation in Section 6.2.1.

Another limitation for the tool’s usefulness is how it approaches test suites: they’re currently assumed to be simply part of the body of a main method. In the interest of taking advantage of existing automated test suites, support for automated testing frameworks such as JUnit would be more ideal. It may be possible to use the APIs of such frameworks to exercise the system under test from a main method as a workaround, but care would need to be taken to ensure that all dynamic observation occurs in the same process thread. SpecTackle currently assumes single-threaded execution.

6.2 Future Work

6.2.1 Constraint-based Inference for Loops

Cleanly addressing inference of invariants for loops is a significant hurdle, and largely remains an open problem. Daikon has been used to infer likely loop invariants by observing invariants at loop heads, but Daikon has some key limitations (e.g., only 3 variables) in addition to only providing likely invariants - the test suite must be robust enough to provide enough counterexamples to incorrectly inferred invariants.

However, one of the foundations of a constraint-based approach like SpecTackle’s is, well, constraining predicates – including loop invariants! The other key foundational concept is how to provide automatic creation of candidates for satisfying
the gathered constraints. For pre- and postconditions, \textit{SpecTackle} uses the candidate inference algorithm as described in Section 2.1.3 as the means of generating predicates from “nothing”.

For loop invariants, it may not be as clear cut. At the head and tail of a loop being analyzed by \textit{SpecTackle}, we will have a set of predicates associated with all variables within scope. We can capture a constraint that a conjunction of this set of predicates must imply the loop invariant, $P$, i.e.,

$$(\text{pred}(v_1) \land \text{pred}(v_2) \land \ldots \land \text{pred}(v_n)) \Rightarrow P$$

There are other logical implications naturally built into any discussion of loop invariants as well (such as $(P \land \neg B) \Rightarrow \Psi$, where $B$ is the guard and $\Psi$ is the loop postcondition); such implications dovetail nicely with \textit{SpecTackle}’s approach, and could be included in the collection of constraints generated at runtime.

But once we have such constraints on a loop invariant, how do we generate a \textit{useful} candidate invariant? Pre- and postconditions had intuitive solutions so that we could build candidates during the solving step based on the constraints themselves. It is not clear that such intuitive solutions exist for loop invariants. Instead, could we take inspiration from \textit{Daikon}’s approach, where there is an extensible pool of invariant patterns that can be tried, based on observations from the test runs? Instead of simply testing if a candidate invariant is satisfied by the variables, as in \textit{Daikon}, this system would test (via a theorem prover, presumably) if the candidate loop invariant satisfies all the constraints observed so far. This would take the place of the creative “human touch” often needed for generating loop invariants. While underdeveloped so far, we believe there is potential in this approach.
6.2.2 Mutable Object Types

For any mutable type allowed, one of the first issues to be tackled is that of how to generate invariants for object fields. It appears that generating constraints for the invariants, in a manner inspired by Rubydust’s type constraints for fields may be adequate for this purpose, so that class invariant constraints for each field are generated at method entry and exit.

This leads us to another interesting area of future work – that of inferring abstract types. SpecTackle currently conducts inference purely over the concrete types directly stated in the program code, and no abstract type is included in inference, even simple abstractions such as calling double values “Money” types. Some work has been performed in inferring abstract types dynamically [7], but that only addresses the inference of simple abstractions over base types. If we allow for object types, it would be especially useful to introduce inference of abstract object types, or at least a means for manual introduction of a mapping from the concrete implementation to the abstract type.

Investigations in this direction will also likely need more work around SpecTackle’s solver engine. The current incarnation is able to use a simple, intuitive mapping (e.g., from double to $\mathbb{R}$), but the introduction of more complex, mutable types warrants more investigation into the abstract mapping used, and indeed, possibly even modifying the solving algorithm. Thus, a more robust SMT engine or automated theorem prover may need incorporated when addressing object types.
Bibliography


URL