Automatic Program Verification and Test Case Generation of Ruby Programs

Loren J. Segal

A thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Applied Science (Software Engineering)
Concordia University
Montréal, Québec, Canada

September 2012

Concordia University
School of Graduate Studies

This is to certify that the thesis prepared

By: Loren J. Segal

Entitled: Automatic Program Verification and Test Case Generation of Ruby Programs

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Software Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair
Dr. S. Bergler

_____ Examiner
Dr. J. Paquet

_____ Examiner
Dr. G. Butler

_____ Co-supervisor
Dr. P. Grogono

_____ Supervisor
Dr. P. Chalin

Approved _____
Chair of Department or Graduate Program Director

_____ 20 _____ _____
Dr. Robin A. L. Drew, Dean
Faculty of Engineering and Computer Science

Abstract

Automatic Program Verification and Test Case Generation of Ruby Programs

Loren J. Segal

The Ruby programming language is typically not seen as a language that can be formally verified. Our research attempts to bridge this gap by introducing novel techniques to annotate Ruby programs with type specifications, contracts, and translate them to statically verifiable components. We introduce a novel tool, *RubyCorrect*, which uses these techniques to perform extended static checking (ESC) on Ruby programs, as well as to generate executable test cases through symbolic execution. These analyses serve to improve code quality and development productivity. We aim to show that Ruby programs can benefit from existing static verification tools and techniques if they are simply made available to Ruby developers.

Chapter 1

Introduction

The Ruby programming language is known for its expressiveness, syntactic malleability, and heavy use of dynamically typed code. As such a dynamic language, writing tools to verify or otherwise audit Ruby programs is considered far more difficult than it is for a statically typed language such as Java, which, unlike Ruby, has many verification tools available to it [Flanagan *et al.*, 2002, Chalin *et al.*, 2008, Havelund and Pressburger, 2000, Huisman and Jacobs, 2000]. With the emergence of Ruby on Rails [Ruby on Rails, 2010], the Ruby language has seen an unprecedented increase in popularity and real world usage, becoming a part of the online presence of many well-established companies such as IBM, Amazon, BBC, Cisco, NASA, and more [Working With Rails, 2010]. As Ruby becomes used for larger and larger applications, the ability to verify a program written in this language becomes a vital step in the development process, and there is a real need for such methodologies and tools.

This thesis investigates the use of static verification and symbolic execution techniques in order to improve the verifiability of Ruby programs. We introduce a novel toolchain called *RubyCorrect* with the two tools, *RubyEsc* and *RubyCaseGen* to take advantage of some of the benefits of these methodologies and enable the verification of Ruby programs.

1.1 Problem Statement

Just like any other programming language, Ruby stands to benefit from automated verification techniques such as static verification and symbolic execution. Both of these techniques have been shown to detect many forms of program faults that programmers can miss. Sadly, these techniques do not currently translate well into dynamically typed languages because programs written in these languages lack critical type information at compile time. Furthermore, there is little effort being made to apply these techniques to such dynamic languages.

Rather than attack this problem head on, the Ruby community instead has promoted Test Driven Development (TDD) in order to avoid such problematic code. However, programming errors are not always caught by manual testing, and can therefore lead to significant program faults. Unfortunately, few tools or methodologies besides TDD and manual code reviews currently exist to catch such errors. The Ruby community is therefore left with programming practices that cannot automatically guarantee program validity to any degree.

1.2 Contributions

The methodologies and tools discussed in this thesis aim to introduce a few novel contributions to the Ruby community, namely:

- A tool (*RubyEsc*) that introduces static verification techniques to the Ruby language and a discussion of static verification methodologies for such dynamically typed languages,
- A tool (*RubyCaseGen*) that allows a user to automatically generate test cases for simple Ruby programs,
- The application of symbolic execution techniques to Ruby programs.

In addition to the above new contributions, we also study techniques for the translation and verification of Ruby programs. Specifically, we look at:

- The use of type annotations to provide static typing in Ruby programs [Furr *et al.*, 2009b, Segal, 2012],
- As well as a comparison of intermediate verification languages (Boogie and Pilar) in order to easily translate Ruby programs into a verifiable form.

1.3 Scope

Ruby offers a vast amount of freedom to the programmer. As such, there are many cases both common and uncommon, where it is not possible to perform analysis on certain Ruby programs with existing techniques. With this in mind, we will be limiting much of the analysis of Ruby programs to exclude the use of any runtime modification of programs (dynamic evaluation or modification of program structure via the `eval()` function, or other means), or at best, include these usages in very simple scenarios.

We also only consider programs that have been annotated with extra type or behavioural information. Although there is much research in the field of type inference for the Ruby language [Furr *et al.*, 2009a, An *et al.*, 2011]—and dynamic languages in general [Anderson and Drossopoulou, 2006, Aycock, 2000, Salib, 2004]—we simplify the process by specifying these types manually. We leave room for future research to apply the work in type inference to our methodologies. Therefore, the techniques and tools used in this thesis will not be applicable to existing programs without modification, although in some cases the modification required is very minor.

The techniques discussed in this thesis build upon the techniques and tools used for other languages. Specifically, we base much of our work upon the Pilar intermediate verification language (IVL) of the Sireum framework [Robby, 2007] used to verify SPARK and Java code. Pilar and Sireum are discussed in Chapter 2. A brief background of all the existing verification techniques in this thesis are also discussed.

Chapter 2

Background

2.1 Static Verification

Static verification is the process of verifying a program without having to execute it. Often in this approach, a program is encoded as a series of simple logical predicates known as *verification conditions* (VCs). The process of encoding a program into these VCs is known as VC generation, or *VCGen*. These VCs are then passed on to a theorem prover (such as Yices, CVC3, Z3, or Isabelle) which determines whether the compound statements are "consistent" or not. One such process of encoding a program as logical statements and passing it off to a prover is known as Extended Static Checking (ESC) [Leino, 1998]. Many tools exist for this type of verification, including ESC/Java and Microsoft's Boogie (which will be discussed in Section 2.8).

Static verification generally relies on the presence of specifications for given methods as a means of expressing their intended behavior. This means that for any program to be verified, it must have a "contract" that specifies its pre and/or post states. Without a contract, a prover can establish *consistency*, but cannot prove *correctness*. Contracts are themselves predicates which are passed to a prover, and are not always trivial to write. The following Java code, annotated with JML (the Java Modeling Language [Leavens *et al.*, 1999, Leavens *et al.*, 2006, Leavens *et al.*, 2011]), illustrates the specification of a `cube()` function:

```
// @requires n > 0;
// @ensures \result == n * n * n;
public int cube(int n) {
   int x = n;
   for (int i = 0; i < 3; i++) x *= n;
   return x;
}
```

Figure 2.1.1: A Java/JML program specifying and implementing a cube function

From the complex implementation of the cube function, it may not be clear to see that there is a defect in the loop. In fact, the cube function as written will compute n^4, not n^3. Running this program through an ESC tool should show that this program is incorrect, thanks to the *ensures* specification in the comments above.

However, even with tool support and well written specifications, ESC is not a perfect solution. One main drawback to this method is that although logical predicates grow linearly with program size [Barnett and Leino, 2005], the execution time of theorem provers does not, and it is very difficult to optimize them. Instead, the common method of improving prover performance is to make use of large server clusters and distribute the work over multiple machines in order to minimize the cost [James *et al.*, 2008], but this requires large amounts of hardware resources.

2.2 Symbolic Execution

Symbolic execution is a static verification technique that is seen as a generalized form of concrete execution [King, 1976] and runtime verification. By using symbolic execution, we can run a program in a controlled fashion, capturing its state at every point during the program flow. As the name implies, a symbolic execution tool will execute the program with symbolic (rather than *concrete*) values, allowing the tool to specify or even deduce constraints that will cause the program to pass or fail [King, 1976]. This makes it very effective for test case generation as will be detailed in Chapter 7. We use Kiasan (discussed below) in order to provide symbolic execution of our Ruby programs.

One benefit of symbolic execution over ESC is performance. The runtime performance

of a symbolic execution of a program can be close to its *real* execution. Symbolic execution does have performance issues, and does suffer from *state space explosion* and *loop exhaustion*, but recent work in the field has shown promising results and optimizations [Deng *et al.*, 2007b]. In addition, symbolic execution can discover properties of a program without the specifications that must be present for proper ESC; though the presence of specifications can help to constrain and greatly optimize the work done by a symbolic execution tool.

2.3 Contract Based Programming

In contract based programming, also known as *design by contract*, the developer specifies a list of events or states that must occur both before and after a method is executed. The specification could, for instance, require a range bound on an integer value passed into a method, or specify that a method should return a specific value. These two types of specifications are further classified as pre- and post-conditions, each specifying the respective required inputs and ensured outputs. Such specifications are known as *contracts* because they are agreements between the developer and the users of the code about the expected inputs and outputs of a method, as well as any side-effects. The user is *required* to supply valid inputs, and the developer *ensures* that the method will return the specified outputs. We therefore use this concept of contract based programming to validate specified methods in *RubyEsc* as well as in *RubyCaseGen*, to a lesser extent, in order to generate executable versions of these specifications.

2.4 The Ruby Programming Language

Ruby is a multi-paradigm, general purpose, dynamically typed, programming language with a focus on pure class based object orientation and functional programming. Created by Yukhiro "Matz" Matsumoto in 1993, the language takes inspiration from Lisp, Smalltalk, Perl, Eiffel and Ada [Ruby, 2010].

Ruby's power comes from its clean, easy-to-read, syntax. Ruby allows code to be

formatted in many different ways. For instance, to create getters and setters for "attributes" (otherwise known as properties or fields) in the language, a programmer only needs to call one method, **attr_accessor**:

```
class MyClass
  # Creates methods 'foo'/'foo='
  attr_accessor :foo
end
```

Figure 2.4.1: Creating attributes in a Ruby class

```
class MyClass
  def foo; @foo end
  def foo=(v) @foo = v end
end
```

Figure 2.4.2: Creating getters & setters manually

This builtin functionality is not exposed through a keyword, even though it might look like one, but a regular method call. Developers are therefore able to create their own custom "keyword-like" method calls just like the standard library does in order to create tiny domain specific languages within the Ruby syntax. Such power also makes Ruby potentially unpredictable, because any method can be defined on any class to perform similar metaprogramming tasks. The idiom is used commonly in many frameworks and libraries, such as the following Ruby on Rails model (class) declaration:

```
class Person < ActiveRecord::Base
  acts_as_versioned
  belongs_to :store
  has_many :friends
  property :name
end
```

Figure 2.4.3: Metaprogrammed class methods

All of the method calls in the example above are class methods defined by library code. Their behaviour, although fairly understandable at the human level, can be extremely complex, and can modify the class by creating an arbitrary number of new methods, or even create new classes as needed. In order to manage this extra complexity, documenting and verifying this behaviour becomes of greater importance. Ironically, it is this extra complexity that makes it so hard to create tools that can verify Ruby programs without the aid of extra meta-data.

2.5 YARD

YARD (Yay! A Ruby Documentation Tool) is a documentation tool for the Ruby programming language. The project was initially created in 2007 as a replacement for the then-standard RDoc documentation tool. YARD sets out to add meta-data tags to the documentation syntax in the Ruby community; this syntax was popularized by Java via the well known *Javadoc* tool [Kramer, 1999]. Lately, the `@tag` style syntax has made its way to many other languages, seeing widespread use in Objective-C, C, C++, Python, PHP, and Javascript codebases, with many tools to support these documentation strings. Doxygen [van Heesch, 2011] is one of the better known documentation tools that supports meta-data syntax, but unfortunately does not have robust support for Ruby. Furthermore, Ruby has unique properties and syntax that makes it hard for existing multi-language tools to support it fully. Since Ruby can declare new classes or methods at runtime, the tool cannot guarantee that the standard Ruby syntax will be used to create such classes or methods. Moreover, as was just shown, such metaprogramming is very commonly used in real world Ruby libraries and frameworks to create classes or methods in this fashion. Finally, Ruby meta-data should allow for the extra specification of types, since it is a dynamically typed language. This type information is crucial to the application of YARD to static checking, runtime verification and test generation.

```
# Reverses the contents of a String or IO object.
#
# @param [String, #read] contents the contents to reverse
# @return [String] the contents reversed lexically
def reverse(contents)
  contents = contents.read if respond_to? :read
  contents.reverse
end
```

Figure 2.5.1: Ruby documentation of a method containing YARD meta-data tags with type information

The power of embedding meta-data declarations inside of documentation is extremely apparent once it is used. Simply having the information specified in an easy-to-parse manner makes it possible for tools to verify and validate the written documentation for correctness. This is currently supported by tools such as Exstatic [Mount *et al.*, 2004],

and is also one of YARD's future goals. In addition to verifying the documentation itself, meta-data can also be used to formally describe behavioural properties of the program. There are many examples of formal specification being embedded into such documentation strings, especially in the world of Java, through specification languages such as JML [Leavens *et al.*, 2006, Flanagan *et al.*, 2002] (seen briefly above). Using YARD, we can leverage the meta-data syntax to perform similar static and runtime checking in Ruby.

This meta-data is especially important in a dynamic language such as Ruby. Languages such as JML focus on specifying contracts and omit type checking, because they are implemented for a language with compiler-enforced type checking. Ruby has no type specifications, which is often necessary for runtime checks and mandatory for static checking, and therefore adding this information via YARD tags is an important step in allowing for the verification of many Ruby programs. Adding such tags also limits the complexity of metaprogrammed behaviour. As noted in the introduction, we necessarily ignore code that is evaluated at runtime through `eval()`, however we can occasionally "cheat" if YARD specifications are provided. If the overall behaviour provided by the evaluated code is predictable, it is possible to specify this information in a YARD meta-data tag, bypassing the complexity of Ruby's runtime system. Although this reduces the flexibility of the language syntax, developers can decide when and where to specify this information in more detail, still leaving them some flexibility to *incrementally* add meta-data to a program.

2.6 Mirah

Mirah is a statically typed variant of the Ruby programming language. It is a relatively new language, originating in 2008 and originally developed by the maintainers of JRuby (a Ruby interpreter that runs on the JVM). It is an open source project licensed under the Apache 2.0 license. Its goal is to provide the expressive syntax of Ruby with the performance of statically typed object oriented languages like Java. The compiler is mostly a trans-compiler similar to languages like Scala and Fantom [Odersky, 2007, Frank, 2005], translating Ruby into languages supported by the backend. Currently only Java is supported as a target language. Mirah is able to generate

both `.java` and `.class` files from Ruby-like programs. Mirah itself runs on top of JRuby. Because of Mirah's syntactic similarity to Ruby, it is used as a bridge language to convert dynamically typed Ruby (with annotations) into statically typed Java programs which can be verified using existing tools like Sireum framework, discussed in the next section.

Although Mirah retains most of the Ruby syntax, it makes a few adjustments to the core grammar in order to allow for the type declarations needed to generate statically typed programs. These type declarations are only needed for method arguments and the method return type, as Mirah comes with an inference engine that automatically deduces the type of local variables. Another important difference between Mirah and Ruby is that Mirah does not implement Ruby's standard library. Instead, developers are expected to use the standard libraries and native data structures of the target language, though abstractions are being worked on to allow for backend portability between Mirah programs.

2.7 The Sireum Framework

The Sireum framework is a collection of tools developed by the KSU SAnToS team that aids in program verification, from source translation to lower level logic prover and model checking tasks [Robby, 2007]. The system uses a pipes and filters architecture to call on various components in the framework, depending on what kind of verification the end-user needs. Sireum itself is used mainly to perform model checking and symbolic execution, though its functionality is slowly growing towards ESC through the addition of extra components in the pipeline. Although there are many components in the Sireum architecture, this section discusses the two relevant components, Pilar and Kiasan, that are used by the *RubyCaseGen* tool (later discussed in Chapter 7).

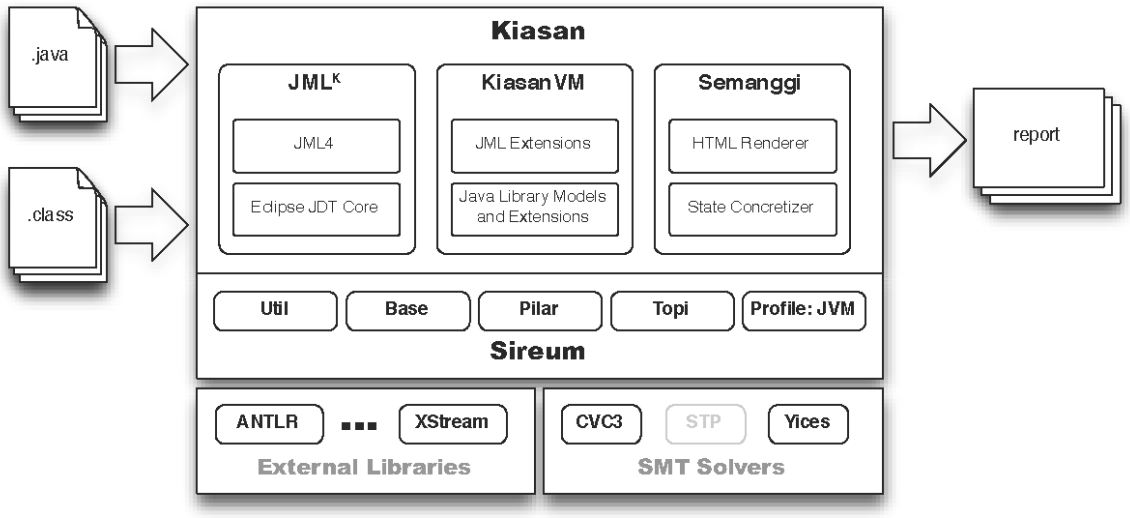

Figure 2.7.1: Kiasan component architecture pipeline within the Sireum framework

2.7.1 Pilar

Pilar is an intermediate verification language used by Sireum to describe source programs in a consistent syntax (and internal structure) across all components in the pipeline. By using Pilar across the framework, any specific component can be easily re-used for other source languages and purposes. We describe the feature set of the Pilar language in much greater detail in Chapter 4, as well as compare its features with alternate IVLs (mainly, Microsoft's Boogie).

2.7.2 Kiasan

Kiasan is the symbolic execution component for the Sireum framework. Pilar is fed into Kiasan and executed symbolically in order to generate the various states for each path executed in a program. These states are then exported as XML and can be analyzed in order to generate test cases. The full pipeline is seen in Figure 2.7.1. Kiasan has separate profiles to support execution in particular environments (virtual machines). Currently, there are profiles to support the execution inside Java [Deng *et al.*, 2006, Deng *et al.*, 2012] and SPARK/ADA [Belt *et al.*, 2011]. Because of the complexity required to adapt a profile to Ruby, and because of the similarities between the JVM and Ruby's object model, we opted to re-use the JVM profile.

This, however, requires us to translate Ruby source into JVM bytecode. Fortunately, as discussed in Section 2.6, there already exist a few tools that can perform this translation in a number of ways.

2.8 Boogie

Boogie is a tool and intermediate verification language created by the RiSE team at Microsoft in order to implement extended static checking, primarily for their .NET family of languages (C# specifically), though it also exists as a backend for tools that verify the C language (HAVOC, VCC) [Microsoft Research, 2012, Cohen *et al.*, 2009]. The tool accepts programs translated into its IVL (also named Boogie) and performs verification condition generation (VCGen) on that given input. Boogie uses Z3, a theorem prover (also developed by Microsoft), to handle the verification conditions and report the satisfiability of the given theorems. Boogie has an extensive syntax and is detailed in Chapter 4.

Chapter 3

The State of Verification and Testing in Dynamic Languages

Next generation high level languages are becoming more and more synonymous with dynamic (and/or dynamically typed) languages through their implicit introspective runtime functionality. Due to this shift of pushing functionality away from compilation to runtime interpreters, less is known about these programs at compile-time, which leads to more problems that can go unchecked prior to execution of the program. This means that:

- it is inherently more difficult to perform static analysis on this class of languages, and,
- runtime (unit) testing gains a more prevalent role in the verification process of these programs.

While research in verifying programs in these languages is attempting to catch up with the level set by research in verifying statically typed languages such as Java, the practice of runtime testing is becoming a dominant part of development processes that make use of dynamic languages. In this chapter, we identify ongoing research in program verification, and analyze data from surveys conducted on testing in these dynamic (and dynamically typed) languages.

3.1 Dynamic Versus Dynamically Typed

It should be clearly stated that this thesis is meant to discuss both *dynamic* and *dynamically typed* languages. We define *dynamic* as the ability of the language to mutate its own program structure at runtime (*e.g.*, add or modify classes, data structures, typing rules, and functions). We define *dynamically typed* as a language that only performs type checking at runtime, if ever. Most *dynamic* languages used today are also *dynamically typed*, although not all *dynamically typed* languages are *dynamic*. In this thesis, we consider (mainly) the languages Python, Ruby, Scheme and JavaScript, all of which are both *dynamic* and *dynamically typed*.

Unless otherwise noted, when we say *dynamic* in this thesis, we refer to languages that are both *dynamic* and *dynamically typed*.

3.2 The Infeasibility Assumption

Although many papers *mention* the infeasibility or difficulty of formal static analysis in dynamic languages [Holkner and Harland, 2009, Furr *et al.*, 2009a], few of them actually attempt such static verification or analysis in these languages. It is generally taken as the "infeasibility assumption" that these languages are poorly suited for such formal verification. Although this assumption is more than likely accurate to some degree, we summarize below research that specifically tackles the problem of verifying the correctness of programs written in various dynamically typed languages.

There is much existing research that is focused on type inference, such as Christopher Anderson's work on inference in JavaScript [Anderson and Drossopoulou, 2006], Aycock's and Salib's work in Python [Aycock, 2000, Salib, 2004], Furr et. al's work in Ruby [Furr *et al.*, 2009b], or most significantly, the *soft typing* paper by Cartwright and Fagan [Cartwright and Fagan, 2004]. Although these works are not technically classified under *verification*, all of these researchers believe that inference is a *means* to static program analysis, since type information is crucial to most verification processes. More importantly, however, these researchers all agree (with Cartwright and Fagan being the most explicit about this ideal through their soft typing proposal) that

programs need not lose their expressiveness in order to gain the optimizations and verifiability of statically typed languages. With this mindset, they believe that verification will not be achievable until a proper type system is devised that is compatible with the existing syntax and semantics of dynamically typed languages.

Due to this focus on "solving the type system first", we have observed a lack of research in the field of dynamic language verification. We believe that if there were less emphasis on maintaining the expressiveness of the dynamic languages in question, there would be more advancements in this field. It should be noted however, that of the researchers named above, Cartwright was the only one who managed to move beyond his work in *soft typing* and actually apply it to program verification [Cartwright and Felleisen, 1996].

3.3 Classifying Dynamic Languages

Much of the problems with type inference in dynamic languages stem around the fact that in addition to being dynamically typed, these languages can also change their entire structure, including their typing rules, at runtime. This means that even if type inference worked perfectly at compile time, it is still not enough to say these programs can be verified unless *all* of their runtime states were also known at compile time. This argument remains a huge problem for proponents of type inference in these languages, as well as those who are interested in using this research for verification.

More recent pragmatic research, however, has shown that there may actually be a class of dynamic programs that *can* be verified fully (both static and runtime states), even though they are written in dynamic languages. As John Aycock argues in his paper *Aggressive Type Inference* [Aycock, 2000], "giving people a dynamically-typed language does not mean that they write dynamically-typed programs" (the same argument would be made for *dynamic* programs). Indeed, research published in the last few years studying the "dynamism" such dynamic languages have shown that although the vast majority of dynamic programs make modifications to their structure at runtime, a much smaller number of these programs actually perform any modification *after* the program is initially loaded in memory. Surveys using data from real programs

in both Python [Holkner and Harland, 2009] and JavaScript [Richards *et al.*, 2010] have echoed similar results. Although these findings are preliminary, the data hints that there may effectively be a more practical classification of programs using these dynamic languages into those that are dynamic after load time, and those that are not. The existence of such a classification is further validated by the fact that there has been some recent successful compiler optimization work for dynamic programming languages that takes advantage of this type of program behaviour, specifically for the JavaScript programming language [Gal *et al.*, 2009]. In any case, such a distinction would allow researchers to focus on the latter as being effectively as verifiable as a program written in a static language. We believe that it would therefore be helpful to see more data in the same vein, for other dynamic programming languages.

3.4 Verification in Dynamic Languages

Although we pointed out that much research is focused on type systems in these dynamic languages, there are a few significant findings directly related to verification. Most of this work is able to skirt the type system by performing verification that is less dependent on type information such as data flow analysis and symbolic execution.

There are quite a few data flow analysis techniques and tools proposed for dynamic languages, specifically with their prolific usage in emerging web applications. Being able to verify these programs prior to deployment is becoming a valid *security* concern, and has plenty of practical application. PHP, for instance, is a dynamic language that has become the target of a few research papers attempting to perform static detection of code that may be vulnerable to common security vulnerabilities such as SQL injection and cross site scripting [Huang *et al.*, 2004, Xie and Aiken, 2006]. It is important to note that both of these researchers relied on annotations to fill in the gaps in the verification, rather than strictly relying on type inference methodologies. As a result of this pragmatic decision, they have achieved more complete verification in their research.

JavaScript, in general, is also becoming the focus of much work in verification, as it is

a dynamic language that has been adopted by almost every web presence due to its prevalent browser support—through the ECMAScript standard [ECMA, 2009]. This popularity has put it in focus for much research, allowing for the application of many verification techniques to this language. Kudzu, a recently developed symbolic execution framework targeted for JavaScript [Saxena *et al.*, 2010], is one of the products of this research. Another focus is on "DHTML" (*Dynamic* HTML), which is the use of JavaScript to generate or manipulate HTML pages in a web browser, which has been the focus of at least one study [Tateishi *et al.*, 2006].

We have found, however, that there is not much research being done in other dynamic languages though we believe that it would be interesting to see such research applied to such languages in order to develop better tools and integration with IDEs, as proposed by [Dolby, 2005].

3.5 Testing in Dynamic Languages

Testing is seen as an important process for programs written in dynamic languages [Marvie, 2008]. This would not be apparent if looking at academic research alone, as there is nearly no research in testing methodologies dedicated to dynamic languages. This can be attributed to a few factors, namely:

- Testing a dynamic program is conceptually the same as testing a static one.
- Testing is usually performed at runtime in the target environment, which poses no problem for dynamic languages, since their state is known and testable at runtime.

We have, however, identified a practical survey on the subject of the *use* of testing programs written in dynamic languages. The work of [Saeed, 2008] specifically addresses the differences between testing in static and dynamic languages according to respondents in industry. Their data reflects the factors we identified above, concluding, "there is no difference found in [validation] methodologies for dynamic and static languages. The only difference is that dynamic languages emphasize more on testing and debugging as compared to static languages". Therefore, except for the degree

to which it is emphasized in practice, research in testing is the same for static and dynamic languages alike.

Chapter 4

Comparison of Intermediate Verification Languages: Boogie and Pilar[1]

4.1 Motivation

In this chapter we compare two intermediate verification languages, Boogie and Pilar, in order to determine which language and associated tools are best suited to support the implementation of *RubyCorrect*. We look at the following criteria to make our decision on which IVL to use:

- the degree of similarity between Ruby and the supported features of each IVL,
- how easy it would be to perform automatic source translation to a given IVL,
- the tooling support (performance, ease of use, maturity) surrounding each IVL.

Section 4.6 contains a discussion of our findings and our selected IVL based on these criteria.

[1]Portions of this chapter were accepted for publication and presented at the 4$^{\text{th}}$ International Conference on Verified Software: Theories, Tools, and Experiments [Segal and Chalin, 2012].

4.2 Comparison of Language Features

4.2.1 Basic Assertion Language

Both Boogie and Pilar have similar assertion languages that can be used to encode verification conditions and be sent to various theorem provers to verify program input. The basic assertion language used in both IVLs can be defined by the simple commands **assert**(*expr*) and **assume**(*expr*). These commands assert the validity of an expression or assume the validity of an expression respectively. The expressions themselves must be boolean expressions comprising of variables, literals, arithmetic, or equality operators. Boogie also specifically allows two logical quantifiers, *forall* and *exists*, though Pilar allows function calls and function types as expression values, which Boogie does not.

In addition to these two basic commands, Boogie adds an extra command to this basic set known as **havoc**, which acts similarly to **assume**, encoding that some *variable* is assumed to now contain some unknown value.

4.2.2 Basic Control Flow

Boogie and Pilar are both, at their core, block based languages. They support control flow and branching through this fundamental concept of blocks. This allows them to model the control flow graph of source languages quite closely. It should be noted that *BoogiePL* (the original version of Boogie) was purely block based and had no abstraction for procedures, which illustrates the fundamental nature of this construct.

4.2.2.1 Location and Blocks

In Boogie, a block refers to a sequence of statements to be executed in order. Every procedure has at least one block, though if a block is not specified at the start of a procedure, Boogie will create an anonymous implicit block. The Boogie procedure in Figure 4.2.1(a) shows two blocks, one implicit, and one explicit. The first two

```
procedure run() {
  var x: int, y: int;
  x := x + 1; // #1
  y := x + y; // #2
subtractX:
  x := x - y; // #3
}
```

(a) Locations in Boogie

```
procedure run() {
  local Integer x, Integer y;
  # x := x + 1; // #1
    y := x + y; // #2
  #subtractX.
    x := x - y; // #3
}
```

(b) Locations in Pilar

Figure 4.2.1: Locations in Boogie and Pilar

statements are part of the implicit block that Boogie adds to the start of the procedure and the last statement is part of the *subtractX* block. If no `goto` statement is provided for a jump, Boogie will automatically jump to the next block in sequence. Therefore, the statements 1, 2 and 3 will be executed in order.

Pilar has the same basic concept of blocks, but they are called "locations". The equivalent of the Boogie example is shown in Figure 4.2.1(b). Pilar requires the explicit declaration of the first location, though it does not need to be named. Finally, as shown, Pilar and Boogie will both implicitly jump to the next block (or location) in the source, if an explicit jump is not provided.

In addition to standard block sequences, Pilar also supports non-deterministic choice through a "choice operator" (explained further in the next subsection), similar to Dijkstra's guarded commands [Dijkstra, 1975], which potentially allows for *parallel execution* of statements. Although this feature is not discussed, it can be useful for the modeling of concurrent systems, or where there is non-deterministic behaviour. It is unclear how Boogie would be able to model similar concurrent (or non-deterministic) systems.

4.2.2.2 Branching and Looping

Both Boogie and Pilar can deal with control flow in terms of unstructured `goto` or `return` statements, which can be placed in any location or block. Boogie, however, has many convenience syntaxes for elements such as if statements and loops, and does

```
var x: int, r: int;
x := 0; r := 0;
while (x < 10) {
  if (x < 5) { r := r + 1; }
  else { r := r + 2; }
  x := x + 1;
}
```

(a) Boogie

```
local Integer x, Integer r;
# x := 0; r := 0;
#loop. :: (x < 10)
     +> goto if;
     | else goto endloop;
#if. :: (x < 5)
     +> r := r + 1; goto endif;
     | else r := r + 2; goto endif;
#endif
  x := x + 1; goto loop;
#endloop
```

(b) Pilar

Figure 4.2.3: while and if statements in Boogie and Pilar

not require `goto` statements or blocks for these. To exemplify the syntax for both languages, consider a Java for-loop with an if statement inside of it:

```
int x, r = 0;
for (x = 0; x < 10; x++) {
  if (x < 5) r = r + 1;
  else r = r + 2;
}
```

Figure 4.2.2: Basic branching and looping syntax in Java

Figure 4.2.3 presents one possible encoding of such a loop into Boogie and Pilar respectively. Boogie resembles the high level Java syntax much more closely and is therefore much more convenient to encode to. Specifically, a translator would not need to keep track of (or even consider) location names as is the case for the Pilar equivalent code. Since most popular languages use structured looping and branching constructs such as if/else and for/while, this significantly simplifies translations.

4.2.3 Annotations

Pilar relies heavily on its `@AnnotationName arguments...` annotation syntax to encode source language-specific constructs. Annotations can be attached to any Pilar node, from method declarations to variable references. As we will see, contracts are specified through annotations by attaching them to method declarations. In this

sense, annotations are a very important part of the language syntax. Even types can be encoded using annotations:

```
procedure inc(x @Type Integer) { # x := x + 1 }
```

However, it seems as though abuse of this annotation syntax can end up delegating too much of a source language's features to individual back-end tools, leading to too much complexity in the back-end tooling. For instance, encoding types as annotations entirely bypasses the inheritance and sub-typing semantics that one would get "for free" by using the `record` keyword to declare a type. It would therefore rarely be recommended to encode types in this manner in Pilar.

Boogie also allows for annotations (though they are called *tool directives*) in the form (where Ref is a reference type):

```
var { :NonNull } x: Ref;
```

Such a variable x would be marked as `NonNull`. The equivalent Pilar would be `MyClass x @NonNull`. Neither of these formats have any semantic meaning in the default language. Tools would have to look for these annotations and encode the semantic meaning themselves, either via another source transformation or a computation.

It is important to note that although Boogie has annotations, they can not be used everywhere, *i.e.*, on assignments, variable references, or control flow syntaxes. This can affect tooling if the tool wishes to use annotations to keep track of source code position information to and from the source and destination languages.

4.2.4 Specification of Contracts

4.2.4.1 Specifying Pre and Post Conditions

Boogie has a special construct for specifying pre and post conditions of a procedure. An example is shown in Figure 4.2.4: multiple `requires` or `ensures` clauses are allowed, and the `old` operator refers to the state of a specific variable in the pre-state

```
procedure inc(x:int)
  returns (r:int)
  requires x >= 0;
  requires x < 100;
  ensures old(x) + 1 == r;
{ r := x + 1; }
```

(a) Boogie

```
procedure inc(Integer x)
    @pre(x >= 0) @pre(x < 100)
    @post(old(x) + 1 == x) {
  # x := x + 1;
  # return x;
}
```

(b) Pilar

Figure 4.2.4: Pre and post conditions in Boogie and Pilar

(before the procedure run).

Pilar has no set syntax for declaring such clauses. In Pilar, one would use annotations to encode pre- and post-conditions and rely on tools to process this information. For instance, if VC generation is performed, it would be the tool's responsibility to check for properly named annotations. Although this allows for more flexibility, it also requires more discipline to ensure that the tools used to translate the source to Pilar are compatible with the tools used to process the generated Pilar code. It would also be the source translation tool's responsibility to insert contracts using the correctly named annotations (as expected by the rest of the tools in the workflow). Therefore, *one* possible Pilar equivalent of the Boogie example is shown beside it in Figure 4.2.4.

4.2.4.2 Specifying Loop Invariants

Loop invariants in Boogie are specified with the `invariant` keyword attached to looping constructs. Again, Pilar uses annotations to define these invariants. An example of a loop invariant in Boogie and Pilar is shown in Figure 4.2.5.

```
while (x<10) invariant y==0; { }
```

(a) Boogie

```
(x<10) @invariant(y==0) +> ...
```

(b) Pilar

Figure 4.2.5: Loop invariants in Boogie and Pilar

4.2.5 Modeling Data Structures and Object Oriented Type Systems

One of the most basic features an IVL should support is the encoding of language specific data structures. Both Pilar and Boogie have relatively different syntactic methods of encoding data structures, though their semantics are roughly the same.

Pilar has a syntactic `record` element which is similar, in a sense, to C's `struct`, and is the singular method of encoding any data structure in the language. A record, like a Java class, can inherit from another and it can also be declared abstract. Boogie, on the other hand, has no construct to represent classes or data structures. One must use the `type` keyword along with `var` declarations to model a structure by defining symbols in a flat namespace which represent the fields.

The flat namespace that Boogie uses is an important difference in the way heap-based structures are modeled between the two languages. Specifically, Boogie gives the user full control over defining how to model "memory allocation", and has no concept of object instantiation. In fact, a considerable amount of detail can be found in the Boogie manual [Leino, 2008] about the ways in which the heap can be encoded. On the other hand, Pilar handles allocations through a `new` keyword. Although Boogie's methodology allows for much more flexibility, it is not exactly clear which languages require this much control over heap modeling. The cost of this flexibility is complexity in modeling object-based systems. For example, Figure 4.2.6 models the field `arrSize` from class `Stack`. However, to do this in a flat namespace, the field must be translated into a variable map of references to values, where *references* are the instances of the `Stack` class, and *values* are of the type defined for `arrSize`. To reference the field data under this scheme, we write `Stack.arrSize[o]` where o is of type *Ref*. We must also have introduced this reference type *Ref*, whereas the Pilar code does not require defining a reference pointer type.

```
var Stack.arrSize: [Ref]int;
```

Figure 4.2.6: A class field member modeled in Boogie

Boogie does not impose any typing rules. They may or may not be specified prior

to static analysis. Again, this makes Boogie more flexible for languages with non-traditional type systems, while Pilar tends to be optimized for OO-based languages. In order to model type relationships of the source language in Boogie, we would require the use of axioms (discussed in Section 4.3.2.2) and an extra supertype declaration as follows:

```
const Object
axiom Stack <: Object;
```

Figure 4.2.7: Modeling class inheritance in Boogie

The code above declares the class *Stack* to be a subclass of *Object*. The same semantics is implicitly defined in our Pilar example, since a record will automatically extend Object (if no explicit superclass is defined). The inheritance syntax of Pilar is like that of Java:

```
record ColorPoint extends Point { }
```

Figure 4.2.8: Modeling class inheritance in Pilar

However, unlike Java, Pilar supports multiple inheritance. Note that Boogie can emulate multiple inheritance by modeling the relationships through independent axioms.

4.2.5.1 Generics Support

Boogie and Pilar both support generics (or "parameterized types") on type declarations. Below is a comparison of how Java generics could be encoded into each IVL. It's important to note, however, that although both languages can encode a generic type, only Boogie performs compile-time type checking on the translated source. That is, only Boogie will raise an error when performing an illegal action such as assigning a value to a variable with the wrong parameterized type, as we see in Figure 4.2.9.

In Pilar, type enforcement is only handled by the specific profile and tooling used; therefore, variable assignments are not guaranteed to be type-safe when passed to the tool.

```
type R t;
var R.data: <a>[R a]a;

procedure m(this: R int)
    modifies R.data; {
  R.data[this] := true;
}
```

(a) Boogie

```
record R<‘a> { ‘a data; }

procedure m(this: R<Integer>) {
  this.data := true; // compiles
}
```

(b) Pilar

Figure 4.2.9: Generics in Boogie and Pilar

In addition to this limitation, Pilar cannot encode complex sub-typing relationships in parameterized types. For instance, the Java syntax `class A<? extends String>` cannot be translated into Pilar's generics syntax. On the other hand, Boogie's type system is flexible enough to specify this kind of a sub-typing relationship.

4.2.5.2 Defining Interfaces

Although neither language has explicit support for interfaces, they can be emulated in both languages. Again we see in Figure 4.2.10 that the Pilar version is much more similar to that of a standard OO language syntax such as Java, except we *extend* the base class rather than *implement* the interface. Note that we are only specifying type relationships here— neither of these declarations add any behaviour that affects method dispatch logic. Such behaviour must be added by the source translation tool, or, in the case of Pilar, optionally through a profile.

```
const IFactory: Type <: Object;
const MyFact: Type <: IFactory;
```

(a) Boogie

```
abstract record IFactory { }
record MyFact extends IFactory { }
```

(b) Pilar

Figure 4.2.10: Interfaces in Boogie and Pilar

4.3 Unique Language Features

Although some of the features below are discussed in a comparative context in this thesis, the following is a brief list of some of the features or goals that are uniquely targeted by each IVL.

4.3.1 Pilar

4.3.1.1 Profiles

Sireum/Pilar introduces an abstracted mechanism for defining the Pilar semantics for a given language. These semantics are encoded in what is called a *profile*. Each profile is specific to a source language and defines behaviours such as method lookup semantics and type checking rules (if any). Note that these behaviours are only applied by other tools in the Sireum framework; external tools would not utilize these profiles. In other words, profiles are tools written specifically for use within Sireum. They are a user-customizable, though Pilar ships with default profiles for Java and Spark. We do not discuss the details of writing a custom profile, as we have no experience with this. Eventually, however, this would be necessary for proper *native* Ruby support inside of Sireum/Kiasan.

4.3.1.2 Pluggable Type System

Pilar does not define any default semantics for type declaration or the use of variables with certain types. Because of this, it is possible to declare variables without any type at all, and therefore express the type systems of many different untyped languages through Pilar's syntax. In addition, by using annotations, Pilar's type system can be expanded to define non-standard type systems. For instance, the annotation `@NonNull` could be used to denote a non-nullable type. It is through the use of custom tooling (or profiles, in the case of Pilar) that these annotations and type declarations can take on meaning.

In many cases introducing types into an IVL is unnecessary, since this information

```
type Any, Ref, Type;
const unique Object: Type;
const unique LinkedNode: Type;
var LinkedNode.data: [Ref]Any;
var LinkedNode.next: [Ref]Ref;
axiom LinkedNode <: Object;
```

(a) Boogie

```
record Stack {
  data; // untyped field
  LinkedNode next;
}
```

(b) Pilar

Figure 4.3.1: Defining a data structure in Boogie and Pilar

is often already checked in the compiler of the source language. This is the reason that Pilar's type system is pluggable. It's useful to note that *BoogiePL* (the previous version of Boogie) had less of a focus on a type system (and had been more or less "untyped"), but has since moved toward a much more complete and strict type system (that does not easily allow for "untyped" declarations).

4.3.1.3 Untyped & Dynamic Type System Support

Pilar has support for untyped and dynamically typed languages because of its pluggable type system (discussed in Section 4.3.1.2). As we saw in Section 4.2.5, a data structure can be defined using the `record` keyword and a list of members. These members can have a type specified, or none at all. Boogie, on the other hand, does not support this form of "untyped" data members, and enforces that all members have a type.

Figure 4.3.1 shows the difference between representing a simple data structure (class) in both languages. There are two main observations to be made from this example. Firstly, the Boogie variant looks slightly more verbose than the Pilar source. This is because Boogie does not immediately impose any restrictions on type definitions. In fact, the definition is actually incomplete, since it does not bound the `next` field to a `LinkedNode` type. This type of checking would only be enforced if explicit checks (in the form of axioms) were added to the Boogie source, adding much more required specification in order to model the same data. Pilar could perform this type checking transparently through the tooling (if enabled). Secondly, we can see that Pilar makes it easier to denote a completely untyped field. The Boogie source defines type `Any`,

```
procedure add(x, y) { # return x + y; }
procedure sub(x, y) { # return x - y; }

procedure applyToOneAndTwo(f) {
  local result;
  # call result := f(1, 2);
  # return result;
}

procedure test() {
  local x, y;
  # call x := applyToOneAndTwo(add);
  # call y := applyToOneAndTwo(sub);
  ...
}
```

Figure 4.3.2: A first-class function/procedure object in Pilar

but it is incomplete, since it does not handle primitive types. Again, an `axiom` would be required to define this semantic in the language[2].

In addition to support for untyped data structures, Pilar also supports untyped procedure arguments, variables, as well as untyped functions. Pilar can call procedures with unknown arguments, effectively allowing for a dynamically typed programming style. An example of this style is shown in the next section.

4.3.1.4 Functional Programming

Pilar can easily translate semantics of languages with first-class functions and procedures because it too has this capability; *i.e.*, functions and procedures are first class citizens. In practice, this means that Pilar can assign procedure objects to variables, as well as pass them as arguments to other procedure calls—*e.g.*, see Figure 4.3.2. Note that although the example does not list the type of `f`, the declaration for a function object would actually look like:

[2]We attempted to test the output of Spec# using the `dynamic` keyword in order to emulate how Boogie would behave for untyped fields, but this keyword seemed to be unsupported by the compiler. It is therefore unclear how this translation would be done by Boogie, or if it can be done at all.

```
(Integer v1 * Integer v2 -> Integer) f;
```

Functions in Pilar can also be defined in a short-hand form. For example, the following creates a function and assigns it to `c`, which can then be called as `c("x")` or `c("y")` (returning 5 or 6 respectively):

```
c := ^{ "x" -> 5, "y" -> 6 };
```

Finally, Pilar can accept lambda functions for an even more inline form of the above:

```
procedure Add1(Integer n) {
  local result;
  # call result := (Integer x => x + 1)(n);
  # return result;
}
```

Boogie does not support this form of first-class functions or lambdas, and therefore cannot (as easily) model this type of behaviour. In Boogie, procedures can only be declared in global space and their names can only be referenced as the first argument of a call statement. Similarly, `function` statements can also only be declared in the toplevel scope and cannot be passed as arguments, assigned to variables, or returned from functions (as they are not objects). As an extra restriction, the body of a Boogie `function` must be a single expression.

4.3.1.5 Method Overloading and Multiple Dispatch

Pilar's built-in profiles for Java and SPARKhave semantics to perform multiple dispatch to overloaded methods by selecting the most appropriate method based on the type information in the given methods signatures. For instance, it can correctly dispatch a method call `equals(x, y)` to the correct method given the definitions in Figure 4.3.3.

```
procedure equals(Integer x, Integer y) {
  # return x == y;
}
procedure equals(Meter x, CentiMeter y) {
  # return x * 100 == y;
}
```

Figure 4.3.3: Method overloading and multiple dispatch in Pilar

Note that Pilar will use sub-typing relations to find the most specific type match. This means that it can only support multiple dispatch on types defined explicitly via the `record` syntax, and not types defined through annotations.

In contrast, Boogie has no support for method overloading and therefore has no semantics to do this kind of dispatch. In Boogie, overloading must be handled by performing name mangling on method names (since method names are unique symbols) and encoding an appropriate means to do the dispatch directly in a Boogie program. This would yield extra overhead in performing translation on languages that have overloading, where method names are not necessarily unique.

4.3.1.6 Exception Handling

Pilar allows for explicit exception handling through the `catch` keyword, which allows control to jump to any location when an exception is thrown anywhere inside the procedure. For instance:

```
procedure error() {
  local x, y, o;
  #start. x := 10 / 0;
  #mid. y := 10 / 0;
  #end. return 0;
  #exception. return -1;
  catch ArithmeticException o
    from start to end goto exception;
}
```

Figure 4.3.4: Native exception handling support in Pilar

The Pilar code in Figure 4.3.4 will catch any division errors raised between the `start` and `end` locations only (and therefore only catch the assignment on x) and jump

to the *exception* location. This exception handling construct is designed to closely mimic the semantics of the Java Virtual Machine (JVM)'s own exception handling behaviour.

Boogie does not have such a construct, but exception handling can be modeled by creating an extra out-variable and checking the state of this variable at each function call. Figure 4.3.5 depicts how exception handling might be done in Boogie. This methodology is used to emulate exceptions in *RubyEsc*'s translation of Ruby to Boogie and is discussed in Section 6.4.2.6. A similar approach is also used to model exceptions in the Eiffel programming language [Tschannen *et al.*, 2011].

```
// setting 'e' to nonzero value will raise an exception
procedure externalFunction() returns (r: int; e: int) { e := 1; }
procedure error() {
  var result: int; var exc: int;
  call result, exc := externalFunction();
  if (exc != 0) { goto exceptionBlock; }
  return;
exceptionBlock:
  // perform some exception handling
}
```

Figure 4.3.5: Emulating exception handling in Boogie

4.3.2 Boogie

4.3.2.1 Defining Mathematical Operators

Boogie has the ability to customize the definition of logical constructs from inside the language. Specifically, it is possible to redefine the meaning of mathematical operators such as $+$, $-$, $*$, $/$, and $\%$ to implement the properties of the source language. For instance, in Java, primitive types have bit-specific precision and are subject to properties such as overflow, underflow, and, for floats, decimal precision.

In Boogie, these properties can be mapped with axioms (seen in Section 4.3.2.2). For example, the Boogie 2 manual defines the semantics of division and modulo of an integer type in Java as follows:

axiom ($\forall$ x:**int**, y:**int** • {x%y}{x/y} $x\%y = x - x/y * y$);
axiom ($\forall$ x:**int**, y:**int** • {x%y}
$(0 < y \Rightarrow 0 \leq x\%y \wedge x\%y < y) \wedge (y < 0 \Rightarrow y < x\%y \wedge x\%y \leq 0)$);

Pilar on the other hand has no syntactic mechanism to give meaning to such operators. The semantics for Pilar operators are only defined by the tools that read in Pilar input. Therefore, while these operators can also be redefined in Pilar, it is more difficult because it requires access to the tools that process the Pilar input instead of just the Pilar input itself.

4.3.2.2 Axioms and Mathematical Quantifiers

Boogie allows the user to define certain expressions that should remain true throughout the execution of a program via the `axiom` keyword. These axioms can be defined either when translating a source file or manually entering proofs, and can use mathematical and logical quantifiers such as `exists` and `forall` to specify certain properties manually.

For example, as shown below, an axiom can be written to specify that there will always exist some x and y integers that sum to the value 22. Note that the $\exists$ symbol is equivalent to using the `exists` keyword, similarly for $\forall$ and `forall`.

axiom $\exists$x:**int**, y:**int** • $x + y = 22$;

Although it is theoretically possible to encode these statements as procedures (as Pilar source would have to do), there are often times when this syntax is more convenient for certain proofs and closer to the source language than the procedural syntaxes. By defining certain properties (such as integer overflow), these axioms can form the basis for creating a profile for a specific source language.

Axioms can also be used to specify properties of the source language itself, such as encoding type inheritance rules (in an OO language) or the definition of certain mathematical operators. Both of these features will be looked at later.

4.3.2.3 Comprehensive Type System

Boogie's type system allows for the specification of type aliases, map types, and parametrized types. Most importantly, all of these type declarations are checked within Boogie, that is, it is not legal to make assignments from one type to another. This is often useful to encode source languages with similar typing restrictions, and offers a good sanity check when performing such translation.

4.4 Implementation Considerations

After comparing the syntax and semantics of the language, it would also be useful to take a step back and compare the current landscape of each project. For instance, there are a few practical considerations to note when using either IVL. Such a decision should be based on tooling support, support for theorem provers, as well as platform support for the tools themselves. Of course, we should also consider which source languages are currently supported by each IVL.

4.4.1 Language Support

Boogie was initially developed as the backend for the Spec# project (a design-by-contract extension of C#), and is therefore highly optimized for C# programs (or Spec# programs), and .NET programs in general. Boogie can directly read MSIL bytecode (compiled .NET programs) in addition to its natural text-based syntax. In addition to C# support, there are a few projects (HAVOC [Microsoft Research, 2012] and vcc [Cohen *et al.*, 2009]) which can translate annotated C into Boogie, and there are translators for other languages such as Dafny [Leino, 2010a] and Chalice [Leino, 2010b].

Pilar is part of the Sireum platform [Robby, 2007], which incorporates multiple tools in order to perform various verification techniques on programs. Pilar currently supports Java through the use of JML annotations as well as the ability to directly read JVM bytecode, which makes it a viable option for a whole host of JVM-based languages (like Groovy [Henry, 2006], Clojure [Hickey, 2008], Scala [Odersky *et al.*, 2008]). In addition, it also supports a contract based subset of Ada called SPARK [Barnes, 2003].

By making use of the Sireum platform, Pilar can be used to perform model checking [Robby *et al.*, 2003], symbolic execution and automated test case generation [Deng *et al.*, 2007a]. Such tools do not exist for Boogie, however Boogie can perform extended static checking, which is not yet possible in the Sireum framework. It is also important to note that although Pilar is used by Sireum, there is currently no builtin support in the framework to accept Pilar input as text and perform computations on the resulting model.

4.4.2 Support for Theorem Provers

Boogie can interface with Z3 [Ramakrishnan and Rehof, 2008], Simplify or SMT-LIB (a format supported by CVC3 [Barrett and Tinelli, 2007], Yices [Dutertre and de Moura, 2006] and others). This means it is usable with many different provers. Sireum (the tooling framework for Pilar) supports Yices and has experimental support for Z3. Support for SMT-LIB would likely be a good idea for Pilar's roadmap, since it would give coverage for Yices, Z3 and others at the same time.

4.4.3 Support for Environments and Platforms

Pilar is built on Java, which makes it runnable on virtually any environment and platform. Pilar is also an open source project under the Eclipse Public License (EPL) and can be modified if it does not function on a target platform.

Boogie is built on top of the .NET framework (using C#) and is also open source, under the Microsoft Public License (Ms-PL). This makes it runnable under Windows environments, and partially under Linux (and OS X) environments through the Mono .NET implementation, but it is not fully supported. As mentioned, Boogie uses Z3 as its default prover, but Z3 is not open source and also has compatibility problems under non-Windows environments (though there are Linux and OS X builds available).

4.5 Related Work

IVLs other than Boogie and Pilar exist. We first mention *FreeBoogie* [Grigore, 2007, Chrząszcz *et al.*, 2009] an open-source implementation of Boogie that is built on Java and therefore has superior multi-platform support to Boogie's .NET codebase. It is licensed under the MIT license, and is fairly actively developed [Grigore, 2009].

Why, both the name of a VCGen-based verification platform and the IVL it uses, supports many of the features discussed in this chapter [Filliâtre and Marché, 2007]. Its tooling is built for Java and C VCGen, with back-end support for many theorem provers including *Isabelle*, for which there is currently experimental support in Boogie [Böhme *et al.*, 2010] but no support in Pilar. *Why* is published under the GPLv2 license, making it open source and easily modifiable, similar to Pilar.

Spec# [Barnett *et al.*, 2005] is the main source language for which Boogie is targeted. The language extends the popular C# .NET language, by adding contracts and a non-null type system, among other features. The Boogie tool has native understanding of C# bytecode in order to directly communicate with the Spec# compiler and IDE. *Spec#* introduces many of the features that Boogie supports, including method contracts, class invariants and field checking.

Dafny [Leino, 2010a] is a high level object-oriented language that has built-in support for code contract specifications. It is developed by the RiSE team, the same team developing Boogie. The goal of this language is to provide the same level of abstraction of a modern object-oriented language while still providing access to the same verification power provided by Boogie. In fact, Dafny sits directly above the IVL, translating and sending the program to Boogie for verification before performing final compilation to C# (and .NET).

4.6 Discussion

Our comparison raised many strengths and weakness of each language and related tools. Recall that our criteria was based on: similarity of features, ease of source translation, and available tooling support.

In terms of similarity of features, we found that Pilar would prove to support much more of Ruby's functionality. Its support for dynamic typing, inherent support for object type systems, and anonymous functions map much more closely to Ruby. This affects the ease of source translation, but only to a certain degree. Although Boogie lacks native support for some of these features, it is still possible to model most of the behaviour of Ruby using Boogie syntax; and there were certain features that Boogie was much better at modeling (such as contracts). We concluded that the level of difficulty in performing automatic source translation across these two languages was similar when judged over the entire set of features that need to be supported.

In the end, the decision came down to tool support and language maturity. We decided that we would use Boogie for *RubyEsc* because it was the only IVL with tooling support for extended static checking. Recall that Kiasan does not yet have support for ESC. We also noticed that Boogie's language specification was much more up to date, and the features had better documentation.

Note that we will still also make indirect use of Pilar and the Sireum framework in *RubyCaseGen*, since it uses Kiasan to perform automatic test case generation, something that Boogie does not support. However, we will not be dealing with Pilar directly, and we will only interact with the tool through JVM bytecode inputs (via Mirah's Ruby to Java translation). In the future it might be possible to consolidate *RubyCorrect* to use the Sireum framework exclusively, once better support for extended static checking is introduced.

Chapter 5

Overview of the *RubyCorrect* Architecture

5.1 Overview

The *RubyCorrect* system contains a set of tools to perform extended static checking and automatic test case generation of Ruby programs. In this chapter, the overall architecture of the *RubyCorrect* system is discussed in order to better understand the workflow and requirements of each tool as well as the interaction between the components that are used. Note that this chapter focuses on the novel additions to the Sireum pipeline (Section 2.7), and information on reused components can be found in Chapter 2.

5.2 Pipeline Architecture

The overall *RubyCorrect* architecture is setup as a *Pipes and Filters* architecture [Meunier, 1995]. This design choice allows easier mixing and matching of third-party components, since these components are usually black box entities. The pipes-and-filters architecture is quite suitable because most operations in each tool can effectively be abstracted as a set of sequential translations on input data (Ruby source) which

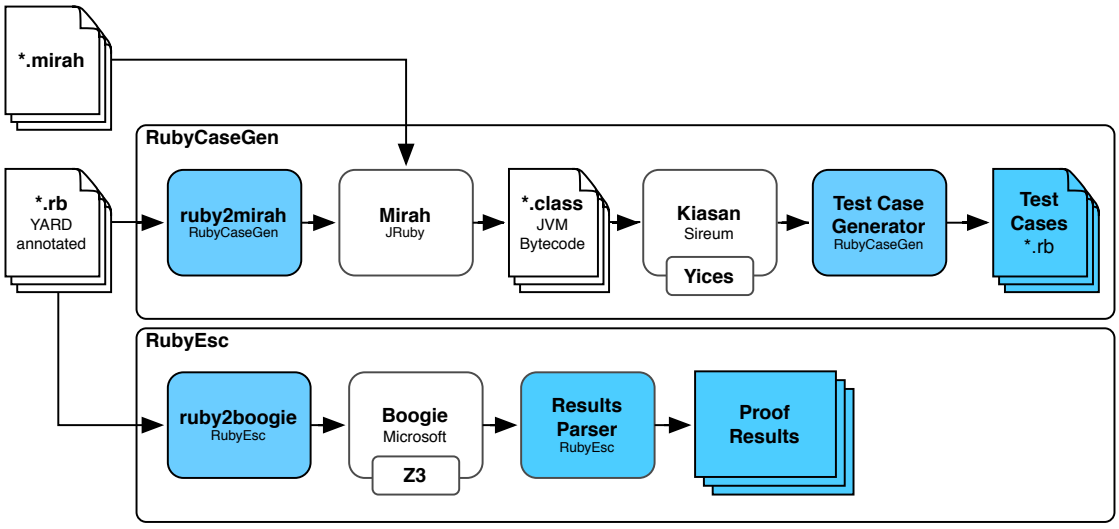

Figure 5.2.1: *RubyCorrect* pipeline architecture overview

is then fed into subsequent tools for analysis (or further manipulation). Figure 5.2.1 illustrates the overall composition of the components in the architecture. The components highlighted in blue are the new components implemented by the *RubyCorrect* system, while the unhighlighted components are existing third-party backend tools that are used to handle verification of the input. The two sub-components are discussed in further detail in the following sections.

The starting point for either of the tools in the *RubyCorrect* suite is a YARD annotated Ruby program. The annotation syntax, as well as required and recommended annotations for use with the tools, are given in the following section. These annotations are necessary to bridge the gap between Ruby's dynamic behaviour and the backend tools, which are typically geared towards statically typed languages.

The annotated Ruby sources are then translated into languages that can be accepted by the backend tools in order to generate results. Note that each tool is effectively a wrapper and glue for the major third-party backend tools. The decision to reuse existing components to handle the proving and verification of the given input was made to minimize the amount of implementation for the *RubyCorrect* system. This is especially important for the *RubyCaseGen* component, which actually relies on translation from Ruby to Java in order to take advantage of the *Kiasan* symbolic execution framework. Rather than implementing support for Ruby within *Kiasan*,

it was decided to use *Mirah*, an existing static variant of the Ruby language, which already compiles to Java bytecode. This design decision is further discussed in Section 5.5.

5.3 Annotating Ruby Code

The major requirement for proper usage of the *RubyCorrect* system is the availability of annotated Ruby code. As mentioned in the introduction, in order to make it feasible for a Ruby program to be fully understood at compile time, it is necessary to provide annotations in the source to document aspects of the program's behaviour. This is especially true when dealing with Ruby's dynamic type system, since the language syntax does not allow for variable type information to be specified, information which is essential for the tooling backends.

In order to provide this information to the backend components, *RubyCorrect* relies on an annotation language embedded into Ruby source files and leveraged during the translation steps of each tool. Fortunately, the annotation syntax is not novel, and is provided by the YARD documentation tool [Segal, 2012]. The basic annotation syntax is embedded inside of Ruby comment lines, similar to the Javadoc annotation syntax [Kramer, 2001], and is illustrated in Figure 5.3.1—the *Fixnum* class represents integers, in Ruby. Only the set of annotation names are new, and are discussed in the following sections. These annotation names are inspired by the annotation syntax of the Java Modeling Language (JML) [Leavens *et al.*, 1999, Leavens *et al.*, 2006, Leavens *et al.*, 2011].

```
# A Ruby comment block
#
# @param [Fixnum] x
# @requires x > 0
# @ensures $result == x * x * x
def cube(x) x * x * x end
```

Figure 5.3.1: Basic YARD annotation syntax inside of Ruby source

5.3.1 Annotating Dynamically Typed Arguments and Variables

In order to provide type information to the backend tools, *RubyCorrect* expects `@param` and `@local` annotations to be defined in front of argument and variable declarations, respectively. For instance, Figure 5.3.2 illustrates the syntax to annotate method arguments and local variables inside of a method. It is important to note that local variables are specified at the method level, not the block level. Although Ruby allows block-local variables, YARD cannot easily understand block-local annotations, and therefore *RubyCorrect* does not currently support multiple block-local variable annotations. In this case, the source should be manually adjusted to use separate variable names for each occurrence of a new variable type in the block. This is also important, since Ruby's dynamic type system allows variables to be redefined under a new type. Such variable redefinitions should also be renamed to new variable types.

```
# @param [String] name
# @local [Fixnum] num_times
def say_hello(name)
  num_times = 5
  num_times.times { puts ‘Hello ’ + name }
end
```

Figure 5.3.2: Annotating method arguments and local variables in Ruby source

The basic syntax for either annotation is specified by the grammar in Figure 5.3.3 in BNF notation. The notation "`.*`" represents any arbitrary text, and "`[A-Za-z_]`" represents a character class of any alpha text (including underscores). Note that `TypeNames` can be a list of types. Ruby allows a variable to contain multiple orthogonal types, but such a specification is currently undefined within our tool.

```
Symbol ::= [A-Za-z_]+;
OptionalDescription ::= .*;
TagName ::= ‘param’ | ‘local’;
TypeName ::= Symbol;
Identifier ::= Symbol;
TypeNames ::= (TypeName ‘,’ TypeNames) | TypeName;
VariableAnnotation ::= ‘@’ TagName ‘[’ TypeNames ‘]’
                           Identifier OptionalDescription;
```

Figure 5.3.3: Grammar rules for *@param* and *@local* annotation tags

5.3.2 Annotating Method Contracts

Method contracts are associated with a method declaration using the `@requires` and `@ensures` clauses for pre- and post-conditions, respectively. The syntax is simply the annotation tag followed by a Ruby expression listing the logical condition. The annotations also allow for three special identifiers to be specified,

- `$result` — which represents the resulting value of the method call,
- `$exception` — which represents a raised exception object (*nil* by default), and,
- `old(`*VARIABLE*`)` — which represents the initial value of a given *VARIABLE* at the start of the method call (prior to any further assignments).

An example of an annotated method contract is given in Figure 5.3.4.

```
# Increases the value of an item in the hash table
#
# @param [String] key the key name
# @param [Fixnum] amount the amount to increase by
# @return [Fixnum] the new amount for a given key
# @requires key.length > 0
# @requires amount > 0
# @ensures old(values[key]) == values[key] - amount
# @ensures $result == values[key]
def increase_value(key, amount)
  values[key] += amount
  values[key]
end
```

Figure 5.3.4: Annotating method contracts in Ruby source

5.3.3 Annotating Class Field Members

Because Ruby is a dynamic language, it also allows for the dynamic creation of class field members, which in Ruby are known as *instance variables*. An instance variable in Ruby is prefixed with a "@", such as `@count`. They are never declared at the class level, and therefore only available at runtime. This means that it is impossible to know which instance variables exist prior to running the program. To deal with this, *RubyCorrect* introduces an `@ivar` annotation with the same syntax as `@param` and `@local`, but defined on the class definition instead of the method declaration.

```
# @ivar [String] name
class Person
  def initialize(name) # constructor
    @name = name
  end
end
```

Figure 5.3.5: Class field member annotation defined on a Ruby class

5.3.3.1 Instance Variables and Attributes

Note that unlike a language like Java, instance variables in Ruby have no "visibility" other than private. In other words, instance variables are never accessible from outside of the class, and must be exposed through method declarations (getters and setters). One such family of convenience methods to declare getters and setters for an instance variable are the `attr_*` methods. These methods declare (at load time) the getter and setter methods on a class. Note that the special method `name=` is called when a Ruby program parses `obj.name = "Name"`, and is therefore the mechanism to handle attribute-like setters inside of the language.

```
class Person
  def name; @name end
  def name=(value) @name = value end
end
```

Figure 5.3.6: Manually defined getter and setter methods in a Ruby program

The manually defined getter and setter methods illustrated above can instead be rewritten using the single line `attr_accessor :name`. The methods `attr_reader` and `attr_writer` could also be used to define readonly or writeonly attributes, respectively.

5.3.4 Annotating Class Invariants

Class invariants in Ruby programs can be defined using the `@invariant` annotation followed by a Ruby expression indicating the property that must be held. Invariants should be defined on the class declaration, in other words, just before the *class* keyword. Figure 5.3.7 shows a *Stack* class with some invariants constraining instance variable (class field member) values.

```
# @ivar [Fixnum] num_elements
# @ivar [Fixnum] max_elements
# @invariant @num_elements >= 0
# @invariant @num_elements < @max_elements
class Stack
  # ...implementation...
end
```

Figure 5.3.7: Annotating method contracts in Ruby source

5.3.5 Annotating Closures and Loops

Loop annotations allow for the translation of loop runtime semantics. The need for loop annotations is not specific to Ruby [Hunt *et al.*, 2006], but because of Ruby's support for closures (discussed below), they serve to denote which blocks might be looping constructs.

It is important to note that support for loop annotations in *RubyCorrect* is simplistic at best, and only the `@invariant` annotation is used to denote loop invariants. Other tools like JML support annotations such as `@assignable` and `@decreasing` [Leavens *et al.*, 2011]; these annotations are not considered in our implementation, but could be supported in future versions.

5.3.5.1 Closure Blocks and Looping

Although Ruby has a standard keyword to denote loop constructs, most Ruby programmers use closure blocks for iteration, allowing anonymous blocks to be called in an iterative fashion. For instance, the *times* method on the *Fixnum* (integer) class allows for a given block to be repeated a number of times. The following example prints "Hello 0" to "Hello 4":

```
5.times {|i| puts "Hello #{i}" }
```

In addition to this primitive looping method, there is also an *each* interface method defined on various core collection classes (*Array*, *Hash*) which perform iteration over their data. Collection iteration is most often seen in the form:

```
array = [1,2,3,4,5]
array.each {|x| puts x }
```

Any method can handle a closure by calling the *yield* keyword with optional arguments to be passed to the block. For this reason, virtually any method call with an associated closure can be a looping construct. This makes it difficult to detect which methods are calling a closure in an iterative fashion, which methods are calling the block exactly one time, and which methods are conditionally calling the block *at most* one time. For reference, Figure 5.3.8 shows how the Array *each* method could be implemented.

```
class Array
  def initialize(*args) @elements = args end

  def each
    @elements.size.times do |i|
      yield(@elements[i])
    end
  end
end

arr = Array.new(1, 2, 3, 4, 5)
arr.each {|x| puts x }
```

Figure 5.3.8: Implementing a looping construct in Ruby

Fortunately, invariants do not discriminate on the amount of iteration performed, be it 0, 1, or 100 times, and so we can annotate any closure block with a potential invariant. Figure 5.3.9 shows how the `@invariant` annotation can be applied to a closure. Similarly to *while* loops, the invariant is asserted at both the beginning and the end of the closure block. Note that just as with class invariants, the annotation simply takes an expression in the context of the given block; that is, block local variables can also be used in the annotation expression.

```
# @local [Fixnum] j
# @return [Fixnum]
def nine_loop
  j = 9
  # @invariant i + j == 9
  (0..10).each do |i|
    j -= 1
  end
  j
end
```

Figure 5.3.9: Annotating a closure in a Ruby program

5.3.5.2 Keyword Looping Constructs

Ruby also has looping constructs built into the syntax of the language, but they are not often used. The main looping keywords are `for`, `while`, and `until` (which acts as a `while` loop with the inverse condition). Figure 5.3.10 shows what each of these looping constructs might look like in a Ruby program. Note that `for` looping takes an iterable object (a class or object that implements the *each* method), similar to the for-each iteration syntax introduced in Java 5 [Flanagan, 2005]. In other words, the `for` syntax is simply syntactic sugar to direct calling of *each* with a closure block, as discussed in the previous section.

```
for x in [1,2,3,4,5]
  puts "Number #{x}"
end

x = 0
while x < 10
  x += 1
  puts "Number #{x}"
end

x = 0
until x == 10
  x += 1
  puts "Number #{x}"
end
```

Figure 5.3.10: Keyword looping constructs in Ruby

Annotating these kinds of loops are no different from closure loops, as they simply are annotated with the proper invariants:

```
j = 9
# @invariant i + j == 9
for i in (0..10)
  j -= 1
end
```

Figure 5.3.11: Annotated *for* loop in a Ruby program

5.3.6 Annotating Frame Conditions

Two annotations, `@modifies` and `@pure` are available to model how a method might modify the heap. The `@modifies` clause takes an instance variable name that is modified by the method, whereas `@pure` specifies that the method performs no modifications to the heap whatsoever (by default we assume that a method will modify the heap in some arbitrary way unless otherwise specified). Figure 5.3.12 and Figure 5.3.13 show examples of these annotations in action.

```
# ... other annotations ...
# @modifies @counter
def increment
  @counter += 1
end
```

Figure 5.3.12: Annotating heap modifications in a Ruby program

```
# ... other annotations ...
# @pure
def get_username
  @username
end
```

Figure 5.3.13: Annotating a *pure* method in a Ruby program

5.3.7 Summary of Annotations

RubyCorrect introduces the annotations, for various contexts, listed in Table 5.3.1. Each context represents the node types where the annotation is valid, in other words, annotations in the class context should be placed as comments in immediately above the class declaration. Annotations can be reused in different contexts, as is the case for the `@invariant` annotation, which is used for both class and loop invariants.

***RubyCorrect* Annotations Summary List**	
Annotation	**Description**
Class Declaration Context	
`@ivar` [Type] name	Annotates the type of an instance variable (field member)
`@invariant` expression	Annotates a class invariant expression
Method Declaration Context	
`@param` [Type] name (*)	Annotates the type of a method parameter
`@local` [Type] name	Annotates the type of a local variable
`@return` [Type] (*)	Annotates the return type of a method
`@modifies` @*ivar*	Specifies that a method modifies an ivar
`@pure`	Specifies that a method will not modify the heap
`@raise` [Type] (*)	Specifies that a method raises an exception of a given type
Loop or Closure Context	
`@invariant` expression	Annotates a loop or closure invariant

(*) Denotes an annotation already supported by YARD

Table 5.3.1: Summary of supported annotations in *RubyCorrect* tools

5.4 The *RubyEsc* Tool

The design of the *RubyEsc* tool mostly takes advantage of the *Boogie* static verifier, which in turn relies on the *Z3* theorem prover. *RubyEsc* simply acts as a conversion pipe to translate annotated Ruby into Boogie syntax and finally parse the results in order to return them to the user.

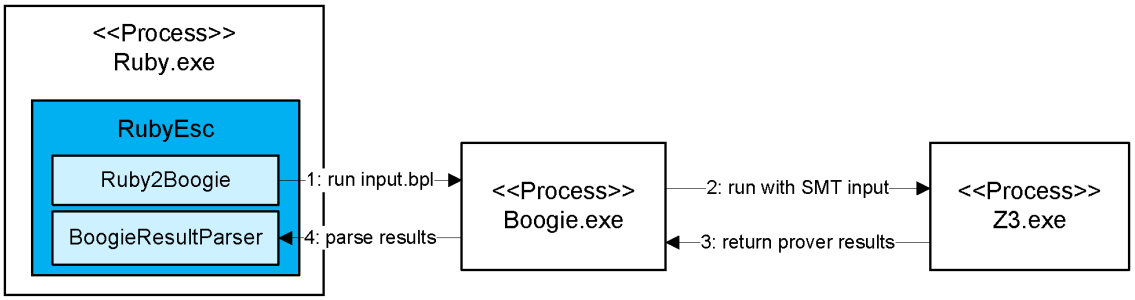

Figure 5.4.1: Process diagram of *RubyEsc* execution

As mentioned earlier, the pipeline architecture allows each subcomponent in the tool to be treated as a black box. We use processes to compose the pipeline. For example, on a Windows system, input data sent to *Boogie.exe* via an file input, and all result parsing is taken from the data sent to standard output when the program completes. The overall *RubyEsc* tool is executed within a Ruby process. Figure 5.4.1 shows the interaction of processes during execution of the tool.

The *Ruby2Boogie* component relies on the YARD runtime Ruby library to parse annotations out of Ruby source while performing translation to Boogie. During translation, the tool maps source locations of nodes in the Ruby program to corresponding locations in the translated Boogie output. These mappings are used to parse the Boogie results back to the original offending locations in the Ruby program, if any errors were reported. This translation, as well as the parsing of results, is further discussed in Chapter 6.

5.5 The *RubyCaseGen* Tool

RubyCaseGen follows the same overall architectural pattern as *RubyEsc*. The main difference is that more steps are needed to translate the annotated Ruby input for the backend analysis tool (Kiasan). Figure 5.5.1 illustrates the overall interaction of components. Specifically, we translate Ruby programs into Mirah, a subset language of Ruby that runs on the JVM, in order to interoperate with the Kiasan symbolic execution framework. Rather than building support for Ruby within Kiasan (feasible, but requiring effort beyond the scope of the prototypical work of this thesis), we leverage existing Ruby dialects in order to use Kiasan without modification. This means that rather than performing a translation from Ruby to Pilar (the input language understood by Kiasan), we instead perform translation from Ruby to Mirah. This translation and the design details regarding Mirah compatibility are discussed further in Chapter 7.

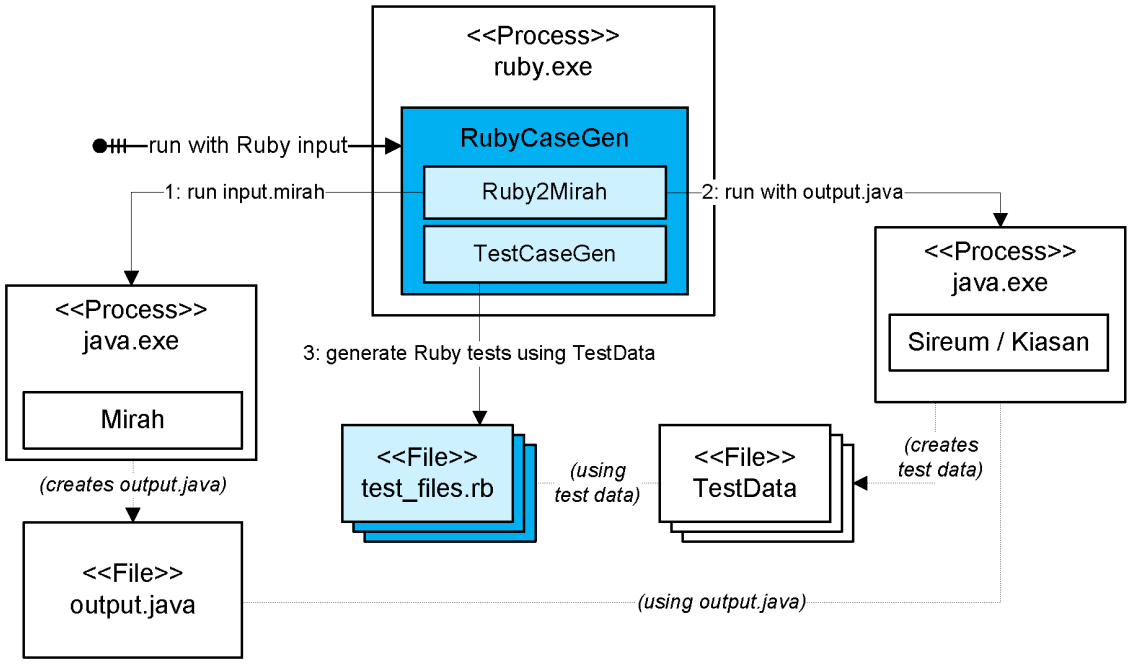

Figure 5.5.1: Process diagram of *RubyCaseGen* execution

Chapter 6

Static Verification of Ruby Programs

6.1 Motivation

Static verification of programs is a useful mechanism to give programmers information about the correctness of their program logic without the need to run the code. Although static languages can provide many standard sanity checks at compile time, this is often not enough to stop many logic errors. Furthermore, in a dynamic language like Ruby, the compiler is even less useful at providing such feedback, and even more errors can go unnoticed in the absence of proper unit testing. We therefore think it is important to target static verification in the Ruby language in order to provide improved early error detection and program correctness.

As mentioned in the introduction, given Ruby's dynamic properties, it would be infeasible to guarantee full correctness over all Ruby programs without modification. We therefore target a subset of the Ruby language and add annotations to fill in any gaps that the language cannot provide automatically to our backend tools. This process is performed as a proof of concept for the general methodology of performing static verification inside of Ruby. We make the assumption that if a basic toolchain and workflow can be created using annotations to aid in the process, further research can then begin to slowly remove many of these training wheels in the form of annotations

until the entire workflow is streamlined to a form that could potentially be adopted as idiomatic Ruby. Realistically, we do not see all annotations disappearing, but developments in type inference can begin to reduce the burden on the programmer to provide annotations; those that are left could be used to form a "best practices" guide to writing verifiable Ruby programs.

6.2 Methodology

This chapter will discuss the overall workflow of writing verifiable Ruby programs using proper annotations as well as the major design and implementation details of the *RubyEsc* tool. In Section 6.3, we provide a detailed listing of the elements in our verifiable subset of the Ruby language, *i.e.*, what Ruby features are and are not supported by the tool. We then provide details for how the annotated Ruby program would be converted into a Boogie specification, discussing the translation of each of the major supported features. Finally, we briefly discuss how *RubyEsc* displays the results from Boogie back to the user, mapping any incorrect or unverifiable statements back to source locations in the original program.

The *RubyEsc* tool was run across various Ruby test programs that contain only our supported features to validate its usability and effectiveness. A brief look at the performance characteristics of the tool is also considered.

6.3 A Verifiable Subset of the Ruby Language

As with most modern programming languages, Ruby's grammar is complex and supports a large collection of syntactic constructs. In order to limit the scope of implementation, we select a small but idiomatic set of features to support. This section outlines the major features that are supported as well as those whose support is explicitly omitted.

Ruby Features Supported by *RubyEsc*	
Keywords	class, module, def, while, until, if, else, elsif, return
Literals	true, false, nil, strings, symbols, array & hash initializers
Method Declarations	Unsupported on instance objects, varargs unsupported
Variable Declarations	**@param**, **@local** annotations used for type specs
Instance Variables	Requires **@ivar** annotation on class for type specification
Global Variables	Unsupported
Arrays	Only 1D arrays supported
Hashes	Unsupported
Constant Declarations	No redefinition supported
Method Calls	Argument count must match declaration
Operators	=, <, >, <=, >=, ==, !=, !, *, +, -, /
Closures	Supported as anonymous method declaration
Looping Constructs	while, until, closures
Expressions	No support for conditionals/loops as expressions
Exception Handling	Basic support for method-level rescue blocks

Table 6.3.1: Summary of features supported by *RubyEsc*

6.3.1 Summary of Supported Features

Table 6.3.1 summarizes the features supported by *RubyEsc* with any relevant caveats. Features and language syntax not mentioned in this table should be considered unsupported.

6.3.2 Known Limitations

6.3.2.1 Logic Outside of Method Declarations

Ruby is considered a *scripting* language in which there is no "main entry point" to the program. Instead, the executed file is parsed top to bottom, and any statements outside of method or closure declarations are immediately executed. Although this is valid Ruby, this behaviour is unsupported by *RubyEsc* and all such statements are simply omitted from verification. Figure 6.3.1 shows an example of omitted statements and the corresponding Boogie output (without prelude). In this example, only a single line, line 4, is translated by our tool. Note that it is possible to collect all unassociated statements and list them within some "main entry point" method since

Boogie does not care about execution order of individual methods, but for simplicity we do not do this. It is also useful to note that the Mirah compiler (used in Chapter 7) will properly move these statements into a traditional *main* method when compiling to Java; this technique could be borrowed.

```
1  # @ivar [Fixnum] x
2  class A
3    # @modifies @x
4    def run; @x = 1 end
5
6    5.times { puts "In class" }
7  end
8
9  if __FILE__ == "test.rb"
10   puts "Hello world"
11 end
```

Figure 6.3.1: Statements omitted when listed outside of method declarations

6.3.2.2 Loading of Files Using `require()` and `load()`

The `require()` and `load()` statements in Ruby will read and evaluate other Ruby source files (or dynamically linked native libraries) from disk. These statements can be placed anywhere in a source file, as they are simple method calls. In general, *RubyEsc* does not support loading of external files. It is, however, possible to check multiple files at a time by listing these files together when calling the checker command. The files will be concatenated together in the order which they appear in the command.

One reason for only supporting multiple files via command line is due to the way loading is performed inside of Ruby. Ruby uses a `$LOAD_PATH` global variable to denote the paths to be searched for a given file, upon a call to `require()`. This variable can be modified at runtime, and it is therefore difficult to know exactly what the state of this variable is upon each call. This variable can also be influenced by system environment changes and command line switches to the Ruby interpreter, creating different initial conditions at each run of the program, all of which cannot be captured in source code alone. Finally, the *RubyGems* package manager library modifies the

behaviour of `require` (see Section 6.3.2.3) in order to load from common system paths for libraries installed as "Ruby gems" (third party libraries). This behaviour is also influenced by environment and configuration settings.

Therefore, explicit annotations would be required to properly know the search paths for calls to `require()`. Instead of forcing the user to specify initial condition state in the source code as annotations, we simply have the user specify the exact files which are to be loaded during course of execution and check those files only.

It should be noted that this methodology also has an extra advantage over directly parsing `require()` calls. Since all files are specified in an opt-in fashion by the user, it is possible for the user to explicitly omit files that are not compatible with *RubyEsc*, or static checking in general (due to lack of type information, code contracts, or due to overly dynamic behaviour). For instance, if an annotated file requires some extra unannotated files, the user would only expect the annotated file to be checked. If `require()` calls are omitted, the user can decide to include these extra files only once they have been annotated and made compatible with the tool.

6.3.2.3 Reopening Classes & Redefining Methods

Similarly to the dynamic behaviour of loading files at runtime seen in Section 6.3.2.2, classes and methods can be "reopened" and redefined (respectively) at runtime. Reopening a class means that the state or methods inside of the class might change after its initial definition by the existence of another `class` construct located in the code.

RubyEsc makes the assumption, based on research cited in Section 3.3, that this dynamic behaviour occurs at "load time" of the program (when files are parsed and runtime modification might occur, but before main execution of the program begins). Based on this assumption, the tool parses through all files in order and takes the final state of all definitions as the definitions that are used during static checking. For example, consider Figure 6.3.2 class *Post* which modifies the *body* method at a later point by reopening the class. This method is redefined from originally returning a *String* to returning a *Fixnum*. Behaviour like this is legal and occasionally occurs inside of Ruby code, although it is not common to change the object type being

returned from a method. In any case, *RubyEsc* will assume that the *body* method of the *Post* class will return a *Fixnum* object throughout the entire execution of the program, even though at one point it was defined differently.

```
class Post
  # @return [String]
  def body; return @body end
end

# ...Other code...

class Post # reopened class
  # @return [Fixnum]
  def body; return 42 end
end
```

Figure 6.3.2: Reopening a class to modify method behaviour

It is worth mentioning that there is a class of bugs that this assumption can miss, namely if "Other code" in the example contains top level code that executes immediately and makes use of *Post*'s *body* method, however we argue that this is not typical of Ruby code, and we simply do not support this practice. It *could* be possible to support this practice by keeping a separate copy of a class at each mutated state, and use the correct version of that class at each respective use. We do not implement this for simplicity sake.

6.3.2.4 Multiple Type Specifications on Variable Declarations

It is possible in Ruby (and other dynamic languages) for a variable container to reference objects of different distinct types throughout the lifetime of the variable. For instance, some variable `myVar` can reference a *String* object at the beginning of a method and then be reassigned to reference a *Fixnum* object by the end of the method. Since the annotations used by *RubyEsc* are provided at the method level, it is not possible to specify a variable that changes its type specification in the middle of a method. Note that YARD does support annotations on any statement, so it could be possible for *RubyEsc* to search for these annotations and know to create a copy of this variable with the new type, but this is currently not supported by the tool.

6.3.2.5 Implicit Return Values of Constructs

In Ruby, there is no distinction between statements and expressions; every statement is also an expression. This applies to any syntax in the language, including class and method declarations. Consider the following example class declaration, which assigns the class object to a variable *klass*:

```
klass = class A; self end
p klass # => A
```

The above code is valid because Ruby always returns the last executed statement of any block as the result of the parent block. In this case, the expression *self* (representing the class object, in that scope) returns the class object A, and, as a terminal statement, is the implicit return value of the `class` block. This same syntax can be used to return the result of `if`/`else` blocks or switch statements. *RubyEsc* does not support implicit returns from any of these blocks.

Note that this same behaviour applies to methods in the Ruby language, where a value is the implicitly returned as the result of the method if it is the last executed statement. Recall that all statements return a value, and there is no such thing as a statement with no return value, though the return value can be *nil*. Consider the method definition:

```
# @param [Fixnum] x
# @return [Fixnum]
def ifswitch(x)
  if x == 5
    100
  else
    0
  end
end
```

Even though there are no *return* statements in this method, it will still return an integer value of either 100 or 0, depending on the input. This is because the last executed statement is always either 100 or 0.

For simplicity, *RubyEsc* does not handle implicit returns in method declarations either, and all returns must be explicit. Although it is not idiomatic to use explicit

return statements in Ruby, it could be possible to support this in future implementations by creating a control-flow graph of each method and detecting terminal statements. Note that if this were done, the return types of each terminal statement must still match, as discussed in Section 6.3.2.4.

6.3.2.6 Standard Library Support

Ruby comes packaged with a large standard library that enables most basic runtime functionality. This standard library is implemented in a mix of native C and Ruby code, and is not annotated with type information, let alone contract specifications. Because *RubyEsc* relies on *annotated* Ruby code, it would be necessary to add annotations for all of these methods, but this is quite an undertaking. For simplicity, we provide only basic annotations for some or the core Ruby classes; these annotated definitions are injected as a prelude to the main Ruby input.

6.4 Converting Annoted Ruby into Boogie

Table 6.4.1 lists the different Boogie nodes implemented by *RubyEsc* with the Ruby constructs that map to each of these nodes. This section describes the process with which annotated Ruby code is translated into these Boogie nodes, highlighting the nontrivial node translations.

6.4.1 Modeling Objects & Classes

In order to understand the basics of translation, we must first discuss how Ruby's object system is translated into Boogie. This translation is unique to a language like Ruby and forms the basis of the control flow and logic translations discussed later.

Ruby Feature(s)	Associated Boogie Node Type
Statements	
assert method call	AssertStatement
variable assignment	AssignmentStatement
assume method call	AssumeStatement
method definition	AxiomStatement
method calls, operator usage	CallStatement
@ensures @requires @modifies annotations	ContractStatement
exception handling	GotoStatement
if statement	IfStatement
exception handling	LabelStatement
return statement	ReturnStatement
initial variable assignment, literals, arrays, fields	VariableStatement
looping and while statement	WhileStatement
Expressions	
simple operator binary expressions	BinaryExpression
instance variable reference	FieldReference
method definition	FunctionExpression
method argument	Parameter
parenthesis usage	ParenthesisExpression
simple unary expressions	UnaryExpression
variable refs, arrays, instance variables, literals	VariableReference

Table 6.4.1: Boogie statement nodes implemented by *RubyEsc*

6.4.1.1 Objects & Integers

Ruby is a high level language with no native types. Although there are optimizations at the implementation level in Ruby's interpreter to deal with integers in an efficient way, integers are mapped as objects of the *Fixnum* class inside of a Ruby program. In practical terms, this means that an integer is just like any other kind of object, and can be assigned to any variable container regardless of its initial type (since Ruby allows for variable re-assignment).

Boogie, however, sees integers as a distinct type, and typically cannot mix with user-defined reference types. Boogie is also *statically typed*, and cannot mix types across variable assignment or method call arguments. Although we know all types in an annotated Ruby program (thanks to required YARD type specifications), this still causes a problem when modeling a typical object-oriented language with reference types and separate native integer types; in short, having two separate object spaces

makes translation much more difficult. In order to use Boogie in a useful fashion, it is necessary to allow integer types and object types to coexist in the same object space while still being able to perform basic math operations that Boogie supports on integers.

Fortunately, Ruby's own reference implementation (*CRuby*) gives us clues on how this can be done, as the details that drive the optimizations on integers can be used in Boogie to model an object system where native integers and references can co-exist in the same space. Ruby's implementation, written in C (as the name *CRuby* implies), uses a special type `VALUE` which is simply a native `long` C type. In other words, the definition of `VALUE` is defined in *ruby.h* as:

```
typedef unsigned long VALUE;
```

This is typical of several language implementations with reference types. What is not typical, however, is how objects and integers share the value space. *CRuby* partitions the `long` value space by using a bit flag at the least significant bit to denote whether the value is an object reference ($lsb = 0$) or an immediate integer value ($lsb = 1$).

We are therefore able to model this value space similarly in Boogie by using a *type alias* from our `VALUE` type to the Boogie `int` type as follows:

```
type VALUE = int;
```

This alias simplifies translation of all Ruby programs, as we no longer need to make distinctions between *Fixnum* objects and other objects when performing translation.

6.4.1.2 The Heap

To properly model object state (member data) in Ruby's object oriented type system, we introduce a global variable `$heap` in all translated Boogie programs. This variable represents a mapping of object references to their respective object value, given a specific field. The actual Boogie definition is as follows:

```
var $heap: [VALUE][field]VALUE;
```

The *field* type is used in order to partition object state into separate buckets by their separate member fields. This allows us to model heap modifications in a more finely grained manner, i.e., at the level of a field modification rather than at the coarse level level of an entire object being modified. Since Boogie requires state modification to be explicitly noted in the *modifies* clause of a procedure declaration, it is important to be able to model this in the most specific manner possible.

6.4.1.3 Methods

Boogie has a similar construct to object-oriented methods known as *procedures*. Although they are in a flat namespace and have no concept of object state, it is easy to model a Ruby method as a Boogie procedure (also using cues from Ruby's reference implementation).

Name mangling. In order to properly map method names to a flat procedure namespace, the name of a translated Boogie procedure is based on the full path to the method in Ruby. For instance, the base procedure name of a Ruby instance method named *replace* inside of a class named *StringStream* would be `StringStream#replace` (the `#` denotes an instance method as opposed to a class method, since Ruby allows for separate instance and class methods with the same name).

We perform further name mangling on a translated method name in order to add the type class of each argument (including the receiver object type) to the final procedure name. This is necessary to deal with polymorphic methods, since the contracts generated for the procedure are always specific to a given receiver class and its arguments (i.e., heap modifications are performed in the context of the class type). Therefore, each method requires a separate procedure definition for each combination of argument types it may receive. For a method with arguments that contain polymorphic types, there may be more than one combination, and therefore more than one generated procedure. In these cases, a copy of the procedure is re-generated with the newly mangled name and the body and contracts are re-rewritten to use these new argument types. Because there may be a large number of argument combinations,

```ruby
class Math
  # @return [Fixnum]
  # @pure
  def zero
    return 0
  end
end
```

(a) Ruby Code

```
procedure Math#zero$Math(self: VALUE)
  returns ($result: VALUE, $exception: VALUE) {
  $exception := $nil;
  $result := 0;
  return;
rescueBlock:
}
```

(b) Translated Boogie

Figure 6.4.1: A simple Ruby method and its Boogie translation

only the initial procedure body is generated automatically; other variants are only generated upon translation of a method call to such a variant.

Out-variables. It should be noted that all methods in Ruby return some value from the execution of a method body. Therefore, all translated Boogie procedures have at least one out-variable to represent this return value. There are no *void* methods in Ruby. We also have another out-variable representing a possible exception object that can be returned from a method if an exception is raised. Details on exception handling will be discussed in Section 6.4.2.6.

Mapping arguments. Finally, all methods contain at least one argument named *self*, denoting the receiving object of the class that the method is defined on. Note that in the case of a class method, this translation remains unchanged, as *self* will contain a reference to the class object itself (since classes are also objects in Ruby). The use of the name *self* is meant to match the equivalent *self* keyword in Ruby which refers to this same object.

Generating equivalent function form. In order for Ruby methods to be accessed inside of contracts, we must also translate methods into a Boogie *function* statement (and respective *axiom*). This process is discussed in Section 6.4.4.1. For simplicity, we omit the function and axiom definitions from all Boogie code listings unless otherwise noted.

Figure Figure 6.4.1 shows an example of a simple Ruby method and its equivalent Boogie procedure.

6.4.1.4 Operators

In Ruby, operators are implemented as methods. This means that although operators are typically used in their standard infix form, they can also be called as regular methods. The following example illustrates how the string concatenation operator can be written equivalently in both infix and method call forms:

```
"string1" + "string2" # valid syntax; operator usage
"string1".+("string2") # valid syntax; using a method call
```

Operators are defined on classes just like any other method, using the `def` keyword and the literal operator name as the method name (unary plus and unary minus operators use different method names since the operator name is in use for their respective binary forms). This means that translation of operator definitions is equivalent to the translation of method definitions.

6.4.1.5 Instance Variables

Instance variables are equivalent to member fields in other object oriented languages, but Boogie has native support of neither of these concepts, so they must be modeled using native Boogie syntax. Fortunately we rely entirely on the modeling of the heap (discussed in Section 6.4.1.2) to implement member fields. Access to instance variables are simply translated as access to heap data. Figure 6.4.2 shows the equivalent Boogie translation of access to an instance variable `@counter` in class A.

```
@counter = 0
c = @counter
```

(a) Ruby Code

```
$heap[self][A$counter] := 0;
c := $heap[self][A$counter];
```

(b) Translated Boogie

Figure 6.4.2: Translation of Ruby instance variables in Boogie

6.4.1.6 Arrays

Arrays are modeled as simple container objects. Array access using the `arr[i]` syntax is an operator, and therefore translated as a standard method call. Because we model

the basic *Array* class functionality in the preamble (see Section 6.4.5), array literals are created via standard object construction and we use the `Array#push` method to load the array with its initial values. All further operations on arrays are also simply method calls.

The only difference with Array objects is that their annotation uses the "container" form, similar to Java generic specifications. Appendix A shows a Ruby program initializing and testing an Array literal alongside its translated Boogie procedure.

6.4.2 Translating Control Flow

6.4.2.1 Basic Control Flow

Ruby is an object-oriented programming language, with methods containing the majority of executable Ruby code (exceptions to this rule are discussed in Section 6.3.2.1). Method bodies contain a single entry point but multiple exit points, and are executed in a sequential fashion from the first to last statement, exiting at any *return* statement or exception. There are more obscure ways to exit control from a block of code in Ruby, but these forms are unsupported.

To model basic control flow, we map a method in Ruby to a *procedure* element in Boogie and translate each Ruby statement as an equivalent Boogie statement inside of the procedure body. Note that as mentioned in Section 6.3.2.5, Ruby sees any statement as an expression, but Boogie has specific delineation about which syntaxes can be used as statements or expressions. One notable difference illustrated in previous sections is the use of if condition blocks as expressions inside of larger statements. As mentioned, we do not support this syntax, and assume that statement syntaxes in Boogie must also be formulated as statements in Ruby code. We also assume that all exit points are clearly marked with *return* statements, even though Ruby does not require this. The statements outlined in Table 6.4.1 show the type of statements that can be found inside of a translated Ruby method. Note that nodes marked as "statements" cannot be used as expressions inside of Ruby.

```
var retval$1: VALUE;
call retval$1, $exception := Object#$eq$Fixnum$Fixnum(5, 6);
assert retval$1 == $true;
```

Figure 6.4.3: Translation of Ruby statement `assert(5 == 6)` into Boogie

6.4.2.2 Method Calls

The most basic functionality of a Ruby program is the method call. Since Ruby is an object-oriented language, most behaviour is translated to a method call on the target object. In order to call a method we rely on the *call assignment* syntax in Boogie. Note that Boogie has two forms of procedure calls, one *simple call* form, and the *call assignment* form. We always use the assignment form, since there is always at least one out-variable, and Boogie requires that procedures with out-variables be called in the assignment form of the syntax with the same number of variables as defined by the procedure.

The translation of a simple statement such as `assert(5 == 6)` would therefore be translated into a method call using the equality operator on two *Fixnum* objects. This translation is illustrated in Figure 6.4.3. The use of the out-variable `$exception` is discussed in Section 6.4.2.6.

In order to find the target procedure of a method call, we perform a lookup based on the type of the target object and the types of the arguments passed in (using the YARD annotated type specifications). The first argument to any method call is always the target object itself, as this is the calling convention discussed in Section 6.4.1.3. The details of this method lookup are discussed in Section 6.4.3.

6.4.2.3 Conditional Branching

Ruby has two keywords for performing conditional branching, *if* and *unless*; the latter performs the negation of the conditional branch. Both of these syntaxes are mapped to an `IfStatement` Boogie node in *RubyEsc* (with the respective negation of the condition). As mentioned above, *if* statements in Ruby code must be used in "statement form", i.e., the resulting value of the *if* block must not be used by any

```
x = 10
if x == 2
  x = 1
else
  x = 0
end
```

(a) Ruby

```
x := 10;
call retval$1, $exception :=
  Object#$eq$Fixnum$Fixnum(x, 2);
if (retval$1 == $true) {
  x := 1;
}
else {
  x := 0;
}
```

(b) Boogie

Figure 6.4.4: Translation of if statement from Ruby to Boogie

other statement or expression, because Boogie does not support this functionality. Figure 6.4.4 illustrates a translation of a Ruby *if* statement into its equivalent Boogie syntax. Recall that infix operators like equality checking are actually method calls, so translation is performed on the condition expression itself, and the resulting out-variable is returned.

Note that if conditions in Boogie must always be of the native type *bool*. Ruby does not have this requirement (any non boolean conditions are simply tested against nil and zero values), so we must translate any non-boolean conditions into boolean ones. In this case, the return value from the method call is a boolean object, but of the Ruby *VALUE* type, not Boogie's native *bool* type. We must therefore convert this condition into a binary expression that tests against the Ruby boolean constant object `$true`. This type of translation is done for any node that is not of type *BinaryExpression*, and is performed in any place where a native *bool* type is required.

6.4.2.4 Closures

Ruby supports anonymous blocks, or closures. This means that a block of extra code can be passed along with any method call, and if the receiving method uses the keyword `yield`, the block of code will be executed from that receiving method. Figure 6.4.5 shows what a simple yield idiom might look like, yielding an integer (n+5) as a parameter to the block of the calling method. As discussed in Section 4.3.1.4, Boogie does not have good native support for this kind of functionality, and so it

```ruby
# executes a block of code
# passing n+5 as an argument to the block
def add5_to(n) yield(n + 5) end

def main
  result = 10
  add5_to(5) do |x| # block body
    puts "#{x} is #{result}" # output '10 is 10'
  end
end
```

Figure 6.4.5: A simple block passed to a method in Ruby

```ruby
# executes a block of code
# passing n+5 as an argument to the block
def add5_to(n) yield(n + 5) end

# create copy method with unrolled block
def main_add5_to(n, resultIN) # pass local state
  x = n + 5
  puts "#{x} is #{resultIN}" # output '10 is 10'
end

def main
  result = 10
  main_add5_to(5, result)
end
```

Figure 6.4.6: After unrolling the block in receiving method

must be emulated.

In order to do this, we create a copy of the receiving method with the closure block unrolled at the point where the `yield` was called. Closures also have access to all local state of the method it was passed from, so we must pass all object state in and out of the closure procedure so that it has read-write access to this state. We name mangle in and out-variables, adding the *$in* and *$out* suffixes to respective variables to avoid variable name collisions with the receiving method. Figure 6.4.6 shows what this might look like if it were done in Ruby.

Finally, because closures are like methods, they can have their own contract annotations (and invariants), so we must also translate all contract annotations, however

these contracts do not make use of *requires* and *ensures* clauses. Instead we perform manual assertions at the start and end of the unrolled block of all pre and post condition contracts. We do this because there might be code after the `yield` call which changes state and could cause the contract to become invalid (similarly for code between the start of the method and the call to yield).

Because this translation is complex, we list the resulting Boogie in Appendix B.

Inlining to avoid contractual islands. One quirk of automatically unrolling a local block into a separate Boogie *procedure* is that procedures in Boogie are treated like contractual islands. This means that when Boogie performs a *call* on a separate procedure, it judges the result not by the actual execution of the procedure, but rather, solely on the outcome specified in the post-condition contracts. In other words, moving code out to a separate procedure might seem like an intuitive way to unroll behaviour, but doing this means that the code is never actually executed alongside the code it was unrolled from. If we increment a counter inside of a block and move this into a separate procedure, Boogie will never know this code was executed unless a post condition on that procedure says it was. This would mean that we would have to specify every behaviour of the closure as a post condition of the method, which is a very difficult translation.

Fortunately we can use a trick in Boogie to get around this "contractual island" problem. By using a special `{:inline}` tool directive on the closure procedure we can have Boogie inline the body of the closure procedure directly into the caller method when it does its own desugaring of the input code.

6.4.2.5 Looping

Ruby has a few different ways to loop, but we only support `while` loops and the `each` idiom. We discuss both methods in this section.

While loops. While loops have a direct analog in Boogie via the *WhileStatement* node, therefore translation for a while loop is straightforward. While loops in Boogie can contain an *invariant* clause that checks whether an expression holds true for the duration of the loop. We can generate an invariant expression by annotating the

while loop with an `@invariant` YARD tag followed by the expression.

Collections enumeration. The `each` enumeration idiom is a slightly more complex looping mechanism. Ruby's *Array* class (and other container classes) implements an `each` method which accepts a block of code (closure) to be executed for each element in the array. For instance, the following Ruby example would print the numbers 1, 2 and 3 to the screen:

```
[1,2,3].each {|i| puts(i) }
```

Fortunately we have a generalized approach to translating closures, and we use the method described in Section 6.4.2.4 to generate unrolled procedures from these closures in our modeled `Array#each` method. The *each* method internally performs a *while* loop over its collection data.

6.4.2.6 Exception Handling

We model raised exceptions much in the same way that we model method return values. Instead of having a single out-variable for the return value of a method, we create a separate special out-variable for the `$exception` variable, which is filled with an *Exception* object when a program calls the `raise` method.

It is important to note that all exceptions in Ruby are unchecked, and therefore we perform no checking of exceptions by default in programs. However, by using the YARD `@raise` annotation on a method definition, we note that calls to that method should check for a raised exception after the call. If the exception check finds an exception, we `goto` a special *rescueBlock* label at the end of every Boogie procedure. If the Ruby method contains a *rescue* block (a method-level try-catch statement), we place the relevant code after this label and set `$exception` to `$nil`. Figure 6.4.7 illustrates what a procedure call looks like if a method contains a `@raise` tag in its specification.

Note that we do not have full support for Ruby exception handling, namely, Ruby allows for multiple rescue blocks with guards for specific exception types. We do not support this checking, and instead only implement the "naked" rescue block. Ruby

```
   call retval$1, $exception := A#might_raise$A(self);
   if ($exception != $nil) { goto rescueBlock; }
   // ...other code...
 rescueBlock:
   $exception := $nil; // if rescue block exists in Ruby method
```

Figure 6.4.7: Boogie translation of a method call to a method that has a `@raise` annotation

also has a more granular `begin ... rescue ... end` syntax which is analogous to Java's try-catch statement. We do not support this syntax either.

6.4.3 Method Call Lookup Semantics

Ruby Lookup Semantics. Since Ruby is dynamically typed, method calls are seen as messages passed to a receiving object (either with the implicit `self` or using the `object.method_name()` notation to pass a message to an object). Ruby interpreters implement their own lookup semantics in order to find the proper method definition associated with that message. The lookup semantics for method resolution in a **Ruby runtime** are listed below. Note that the following list is specific to Ruby's object and inheritance model, and is listed for comparison to our lookup techniques:

- Search in methods defined directly in **receiver's singleton class**,
- Search in methods defined in **modules mixed into receiver's singleton class**,
- Search in methods of **receiver's class**,
- Search in methods defined in **modules mixed into receiver's class**,
- Search in methods of **superclass (and its mixins)**,
- If nothing is found, **call `method_missing`** on the receiver (performing new lookup),
- Default `method_missing` implementation will **raise *NoMethodError***.

***RubyEsc* Lookup Semantics**. We do not implement the complete lookup semantics in *RubyEsc*. Specifically, our lookup semantics omit lookups on the singleton class, as YARD is unable to keep track of annotations on methods defined in singleton classes. For this reason, we do not discuss the details of the singleton class. We also do not perform a separate lookup on `method_missing`, since we have not modeled this behaviour in our preamble.

The lookup semantics performed by *RubyEsc* are as follows:

- Search in **receiver's class**,
- Search in **inheritance chain** of receiver (superclasses and mixins together),
- If nothing is found, **abort translation with lookup error**.

Although the semantics are simpler, we capture the *core* behaviour of Ruby method lookup semantics. In order to determine the *class* of a receiver, we use the type annotations provided by YARD. YARD is also used to perform lookups by class name and in the inheritance tree (which YARD has modeled in its own object graph).

Boogie procedure lookup. Upon performing method lookup, we also perform a search for the translated Boogie procedure given the receiver and argument types. As discussed in Section 6.4.1.3, we generate a separate procedure for each combination of argument types passed in via method call. If this procedure does not exist, we generate a new procedure node on the fly with the proper types.

6.4.4 Translating Contracts

6.4.4.1 Generation of Functions and Axioms

We noted earlier that one problem with translation of Ruby code into contracts is that Boogie contract expressions cannot contain procedure calls or perform any state modification in the same way that procedure bodies can. This is problematic since almost every Ruby operation is a method call, and all methods are translated to procedures.

In order to get around this limitation, we translate every Boogie procedure into a set

```
# @return [Fixnum]
# @ensures $result + 5 == 10
def main
  return 5
end
```

(a) Ruby method with simple arithmetic post-condition

```
function $fn.main(self: VALUE) returns ($result: VALUE);
function $fn.main.exc(self: VALUE) returns ($exc: VALUE);
axiom (forall self: VALUE :: $fn.Object#eq(
    $fn.Fixnum#add($fn.main(self), 5), 10) == $true);
procedure main(self: VALUE) returns ($result: VALUE, $exc: VALUE)
    ensures $fn.Object#eq($fn.Fixnum#add($result,5), 10) == $true;
{
  $result := 5; return;
}
```

(b) Generated Boogie

Figure 6.4.8: Generated function and axioms for a given Ruby method with contracts

of equivalent functions with a `$fn.` prefix in its name. However, functions cannot contain *requires* or *ensures* clauses, and instead define their behaviour using an *axiom* statement. We therefore also translate the pre and post conditions of each method into axiom statements that reason about the function form of the method. This allows us to translate any method call within the contract specifications of a method into a function expression in the Boogie output, rather than a call statement. Figure 6.4.8 shows the translation of a method whose post condition performs some basic integer math and equality testing (name mangling and exception translation removed for clarity).

6.4.4.2 Special Contract Expressions

In order to support the underlying functionality of Boogie contracts, we introduce a few special variables and functions that can be used inside of contracts:

- **`$result`** — the value returned by the method call,
- **`$exception`** — the exception object raised by a method call,

- `old(variable)` — the previous value of *variable* prior to method execution.

We also introduce a special annotation syntax that allows us to embed Boogie code directly into an annotation. This syntax is discussed in Section 6.4.5, since it is only meant for internal usage in the preamble.

6.4.4.3 Pre & Post Conditions

As shown in previous examples, we use the Boogie *requires* and *ensures* clauses to map pre- and post-conditions specified by `@requires` and `@ensures` annotations on a method respectively. The translation uses the function form of a method call, as illustrated in Figure 6.4.8.

6.4.4.4 Invariants

Invariants are only supported on *while* loops, both in Boogie and in *RubyEsc*. In such cases, loops are annotated with an `@invariant` tag followed by an expression, similar to pre- and post-condition annotations. This tag is also used when looping with the `each` enumeration method and a closure block.

6.4.4.5 Frame Conditions

Boogie allows us to model which parts of the program state a method can modify, and these specifications are known as frame conditions. To specify our frame conditions, we use the *modifies* clause attached to the procedure definition. Boogie actually requires a *modifies* clause whenever a procedure performs assignment on global variables (like our `$heap` variable discussed in Section 6.4.1.2). This means that most methods require some sort of modifies clause.

By default we assume that all methods will modify the heap. This is done to satisfy Boogie's strict checking on state modification. We therefore specify the following modifies clause on all methods with no `@modifies` annotation:

```
modifies $heap;
```

This is a very general frame condition. It basically says that this procedure can change any heap object. In order for Boogie to reason about our program in a useful way, we need to provide more specific conditions.

Declaring mutable fields. Using the `@modifies` clause with an instance variable name, we can declare that a given method will perform mutations only on the specified field members. However, these specific frame conditions are not modeled with *modifies*. Instead we use a technique described in Boogie's own technical manual [Leino, 2008] and provide a special post condition guaranteeing that only the specified fields in the heap were modified. For example, given a set of instance variables **var1** and **var2** in class **A**, we specify a frame condition that guarantees only these fields were modified:

$$\begin{aligned}\textbf{ensures } (\forall\ o{:}\textbf{VALUE},\ f{:}\textbf{field} \bullet \$heap[o][f] = old(\$heap[o][f]) \vee \\ (o = \textbf{self} \wedge (f = A\$var1 \vee f = A\$var2)))\end{aligned}$$

This post condition is applied to each method with at least one `@modifies` clause.

Pure methods. Some methods perform no modification to the heap. To specify this, we use a `@pure` annotation. Methods specified as being pure will omit the *modifies* clause in their procedure specification.

6.4.5 Preamble

As discussed in Section 6.3.2.6, we do not model the entire Ruby standard library, but we do cover a small subset of the *Object*, *Fixnum*, *Array*, *NilClass* and boolean classes. These modeled classes and methods are annotated in a preamble Ruby source file which is parsed with YARD along with the input file. In order to avoid needless translation of these methods to Boogie, we use an extra annotation tag `@core` to specify that methods in these core classes should only be generated if they are called by something in the Ruby program. The preamble file is provided in Appendix C.

Because we are modeling core data types, we must occasionally generate contracts that use pure Boogie syntax. To do this, we introduce a special syntax to contract annotations: if an annotation is parsed as a single String literal value (wrapped in

double quotes), it is treated as literal Boogie code and not translated. This special syntax provides an easy way to forego translation of Ruby code in contracts.

6.5 Parsing Results

In order to return useful results back to the user (location of problem in Ruby code), we keep track of every Ruby code location in the original code. We associate every generated Boogie node (c.f. Table 6.4.1) with the closest related Ruby node. Then, once the Boogie output is generated, we keep track of the location (line and column) of each Boogie node. When Boogie reports an error, it prints a line and column which is used to perform a reverse search for the resulting Ruby node. This node is then printed back to the user as the source of the failure. Figure 6.5.1 shows sample output in the case of a verification failure.

```
$ ruby_correct esc examples/ruby_esc/example1.rb
Verification Errors (2):

- A postcondition might not hold on this return path:13:
     return x * x

- This is the postcondition that might not hold:
     @ensures $result == x * x * x
```

Figure 6.5.1: Sample output for *RubyEsc*

6.6 Validation Using Sample Code

In order to validate our translation, we generated sample Ruby contracts and code, packaged into individual "experiments", that cover the basic features discussed in this chapter. Each experiment is placed in a Ruby file that is processed via the *RubyEsc* tool; these programs are all listed in Appendix D. We measure the line of code count (LOC), annotation count (ANN), the number of VCs successfully verified by Boogie ($VERF$), the number of logic errors discovered by Boogie (ERR), the

FILE	LOC	ANN	VERF	ERR	EERR	PTIME (sec)	RTIME (sec)
array.rb	46	1	3	0	0	0.28	1.51
boolean.rb	8	0	1	1	1	0.05	0.95
class_methods.rb	8	1	3	0	0	0.05	1.00
closures.rb	10	2	2	0	0	0.05	1.06
condition.rb	9	1	1	0	0	0.05	1.03
contracts.rb	4	4	2	0	0	0.05	0.98
dispatch_lookup.rb	19	6	9	0	0	0.08	1.06
dynamic_dispatch.rb	22	13	7	0	0	0.08	1.05
equality.rb	9	1	1	3	3	0.05	1.05
exception.rb	17	8	4	0	0	0.08	1.06
exception2.rb	19	8	5	0	0	0.07	1.19
literals.rb	7	1	1	0	0	0.05	1.06
looping.rb	17	4	1	2	0	0.11	1.35
math.rb	6	7	0	2	2	0.06	1.24
operators.rb	9	1	1	0	0	0.06	1.03
stack.rb	18	7	4	1	1	0.08	1.23

Legend

ANN	Annotation count	EERR	Expected error count
VERF	Correct verification condition count	PTIME	Ruby2Boogie parsing time
ERR	Incorrect verification condition count	RTIME	`boogie.exe` execution time

Table 6.6.1: Results of *RubyEsc* across various Ruby example files

number of *expected* errors[1] (*EERR*), time taken to convert the Ruby code into Boogie (*PTIME*), and the time taken to run Boogie on the resulting Boogie code (*RTIME*). We measured the results of 16 total files on a machine with a 3.30Ghz Intel® Core™ i5-2500k CPU running Windows 7 and Boogie 2.2.40414.0705. The resulting data is listed in Table 6.6.1. We note that in some cases, logic errors were expected to be discovered by Boogie, as we were testing various programs with both correct and incorrect specifications.

The example files range from tests on simple Ruby expressions (such as integer math, boolean logic and the use of literal values) to more complex tests on contract translation, dynamic dispatch, array modeling, exception handling, looping and instance variable modeling with frame conditions.

In order to validate these examples, we created an integration test suite that runs the Ruby files, as well as some simpler Ruby expressions to test basic language translation

[1]Expected errors are logic errors that are intentionally introduced into the test programs to exercise Boogie's ability to detect invalid programs, *i.e.*, implementations that do not match a post-condition.

support. Our tests either check that the resulting verification is successful (when EERR= 0), or that the incorrect Ruby specification line exists in the error output from the tool (when EERR> 0). In total, our test suite runs 24 examples in an average of 26 seconds.

In some cases, the number of errors that Boogie reports does not match the number of errors expected from the input program. These results represent translations that are not properly supported inside of *RubyEsc*. We specifically note the inconsistency in *looping.rb*, in which Boogie returns 2 program errors when we expected 0. This is due to a difficulty in specifying a proper loop invariant that Boogie requires for verification; the translation occurs as expected, but the invariant is incorrect.

The results show that *RubyEsc* can effectively verify a subset of small Ruby programs that make use of supported features, namely, simple array handling (push, pop, size checking), integer arithmetic, dynamic dispatch, contract specification, control flow, operator support, exceptions, and closures. The programs are small, but they exercise the exact features that the tool supports in specially crafted tests. Future work would attempt to test our implementation across larger and more complex programs, but this would require a more complete model of Ruby's standard library. This future work, as well as general conclusions about the effectiveness of how *RubyEsc* could be used in Ruby, are further discussed in Chapter 9.

Chapter 7

Automatic Test Case Generation Using Symbolic Execution

7.1 Motivation

Although extended static checking can be of use in any programming language, the barrier to entry for this level of verification is much higher than the alternatives (as will be discussed in Section 9.1). That is, generally speaking, a developer must annotate all methods in a program with contracts in order to get some benefit of ESC. We therefore want to take a multi-pronged approach to providing verification of Ruby code. Symbolic execution [King, 1976] through Kiasan offers a path to this goal with less up-front developer investment. Specifically we want a tool that does not *require* the developer to write code contracts in order for the tool to produce useful results. It also allows us to explore a very different tangible result: instead of a *true or false* response from a tool like *RubyEsc*, we can instead receive snapshots of the state of our program when it failed, giving us insight into the caused of our program failure in ways that we may have been unaware of. In short, while ESC is used to *confirm "pre-existing" expectations of a program*, symbolic execution can be used to **discover what we may not know about our program**. This information can then be used to generate test cases (through our tool) and provide **extra quality assurance** and **productivity** throughout a program's lifecycle.

We have also studied symbolic execution because we believe it is a much closer fit for verification of Ruby programs, and dynamic languages in general. The concept of symbolic execution allows us to inspect real state changes in an intuitive manner and theoretically could allow us to model a much more complete snapshot of Ruby's dynamic runtime behaviour which existing ESC tools could not easily achieve. In other words, we believe that the potential for a verification system based on symbolic execution is much greater than that for ESC, at least for dynamic languages.

In order to explore this potential, we created *RubyCaseGen* as a prototype of a symbolic execution engine for Ruby. We admit that building a full model of Ruby's runtime in a symbolic execution engine would be an extremely arduous process, and therefore we decide to piggyback off of other technologies (namely Mirah, which generates statically typed Java code) to create a simple working tool that verifies a working subset of the Ruby language. We discuss, in detail, the techniques used to build the tool using these technologies, as well as a look at optimizations performed on the generated test cases. We also discuss the limitations of this verification process. Finally, we show results of the experiments performed on various Ruby programs using *RubyCaseGen*.

7.2 *RubyCaseGen* Pipeline Description

This section outlines the basic pipeline for running a Ruby program through *RubyCaseGen*. In short, we perform the following steps automatically:

1. translate annotated Ruby to Mirah,
2. decorate the Mirah program with extra constraints from code contracts,
3. compile Mirah program to a JVM bytecode file,
4. run Sireum/Kiasan on the resulting bytecode to generate symbolic state reports,
5. resolve symbolic values in the Kiasan result files, and
6. generate Ruby output in the form of executable test cases based on the concrete state values.

```ruby
# @param [String] host
# @param [Fixnum] port
# @return [Connection]
def connect(host, port) end
```

(a) Ruby

```
def connect(host:String,
            port:int):Connection
end
```

(b) Mirah

Figure 7.2.1: A comparison of equivalent Ruby and Mirah method declarations

7.2.1 Converting Annotated Ruby into Mirah

Because the goal is to create a tool that tests Ruby code and not Mirah code, we want to be able to convert annotated Ruby code into equivalent Mirah code. As mentioned in Section 2.6, Mirah syntax is very similar to Ruby with the exception of type declarations in the method signature. Figure 7.2.1 shows a comparison of a Ruby method signature with its equivalent Mirah version. It is useful to note that as discussed in the background section on the language, Mirah uses the standard library of the backend language (Java, in our case), and therefore the *String* class refers to that found in Java's core library instead of the Ruby String class. We also translate the Ruby *Fixnum* class to the native Java `int` type.

Our conversion tool is run as the first step of execution of *RubyCaseGen* in order to generate a `program.mirah` file that is then compiled by Mirah. Conversion is fairly straightforward, and simply replaces the argument names (and end of line) with the type specifications found in the `@param` and `@return` annotations of the corresponding arguments. We can therefore perform test case generation via Mirah without the user ever directly interacting with the language.

7.2.2 Adding Contract Annotations

In addition to adding type specifications to the Mirah method declarations, pre-condition annotations are also decorated in method bodies using Kiasan *assume* statements. For all pre-conditions annotated via `@requires` clauses, we place a

```ruby
class Fibonacci
  # @param [Fixnum] n
  # @return [Fixnum]
  # @requires n >= 0
  def fib(n)
    if n < 2
      n
    else
      fib(n - 1) + fib(n - 2)
    end
  end
end
```

(a) Ruby

```
import org.sireum.{...}.Kernel

class Fibonacci
  def fib(n:int):int
    Kernel.assumeTrue(n >= 0)
    if n < 2
      n
    else
      fib(n - 1) + fib(n - 2)
    end
  end
end
```

(b) Mirah

Figure 7.2.2: Mirah translation of a Ruby Fibonacci implementation

`Kernel.assumeTrue(expr)` statement at the top of the method[1]. This is possible because Mirah is a JVM language and can interoperate with the Kiasan library. Kiasan uses these *assume* statements in the bytecode to generate constraints on data types when it performs symbolic execution.

Note that contract specification for symbolic execution adds extra constraint information to the underlying engine (e.g. which numeric ranges might be invalid for a given parameter), but **these constraints are completely optional** to Kiasan. This means that unlike *RubyEsc*, it is not a requirement to provide pre or post-condition annotations in order to use *RubyCaseGen*.

7.2.3 Compiling the Mirah Program

After conversion and decoration of constraints, the Mirah program is passed off to the `mirah` binary which outputs a `.class` file using the same base filename, similar to Java compilation. This step is completely implemented by Mirah itself, we simply initiate the compilation step and wait for the resulting output file. Figure 7.2.2 shows the resulting Mirah translation of a complete Fibonacci method implementation in Ruby.

[1]The fully qualified method is `org.sireum.kiasan.profile.jvm.extension.Kernel.assumeTrue`. This method made accessible using Mirah's `import` statement.

It is important to also note that we do not deal with `@ensures` clauses in our contract translation. This is because *RubyCaseGen* is not meant as a code contract verification tool, but rather as a bug finding tool. As mentioned, the motivation is to lessen the burden on the developer and be able to test Ruby code with as few modifications as possible. We therefore only *optionally* rely on `@requires` and solely to provide constraint hints to the symbolic execution engine (in order to generate more accurate test cases).

7.2.4 Running Kiasan

Kiasan relies on the JVM platform, and the system must have Java installed in order to run this portion of the tool. We execute Kiasan by running Java using `org.sireum.KiasanVM` as the main class and passing 3 arguments to the program: the class name containing a method, the method name to verify, and the *method descriptor*[2] for the method. These 3 values are user supplied, although *RubyCaseGen* can attempt to auto-generate the method descriptor value using the Ruby annotations if it is not provided on the command-line.

7.2.5 Resolving Symbolic Values

Kiasan generates a set of numbered *N-symcase.xml* files which contain the basic structure of a method's state before and after the method is executed (for a series of initial states); however, all of the values are unresolved symbolic values. In order to resolve these values to concrete usable ones, we call on Kiasan again to perform the resolution. Kiasan will internally call on the Yices theorem prover to resolve these values and generate an equivalent object structure which is exported as a sibling XML document under the names *N-testcase.xml*. This XML document contains the concrete values in the method's pre and post execution states and can be used to generate test cases.

[2]The method descriptor is a JVM specific value that represents the parameters and return type of the method signature [Liang, 1999].

7.2.6 Generating Ruby Test Cases

In order to generate test cases formatted as Ruby code, we read the series of *N-testcase.xml* documents (each document is translated to one test case) and use only the *pre-execution* state to recreate the initial conditions of the method and then call the method that is tested by the tool. The test cases generated the standard library *test_unit*, which has a structure and syntax similar to Java's JUnit: a test class is created by subclassing the main test case class, and all implemented methods beginning with `test` are executed with various optional assertions performed on the results.

7.3 Example Usage

RubyCaseGen is designed to test a single method of a class at a time. Therefore, the parameters to the tool are the class name followed by the method name. Optionally, the JVM method descriptor can be passed as the third parameter if the tool is unable to automatically detect the method signature (discussed in Section 7.2.4). The output of the tool is the Ruby executable code that tests the input program—the code can be piped directly into a Ruby interpreter to execute the tests. Figure 7.3.1 illustrates sample output of the tool for a class implementing the Fibonacci series (debugging output and some test cases omitted) followed by the execution of these tests when piped into a Ruby process.

7.4 Techniques & Optimizations

In addition to the basic translation, Kiasan execution, and generated Ruby test code, we perform a set of extra operations on the data in order to provide more optimal results on the input programs. We outline our extra techniques and optimizations in this section.

```
$ ruby_correct case_gen Fibonacci fib
require 'test/unit'
require 'examples/ruby_case_gen/Fibonacci'

class TestFibonacci < Test::Unit::TestCase
  # omitted tests for n=0..3

  def test_4
    this = Fibonacci.new
    n = 4
    result = this.fib(n)
    assert_equal 3, result
  end
end

$ ruby_correct case_gen Fibonacci fib | ruby
# Running tests:

....

Finished tests in 0.00301s, 1332.889 tests/s, 1332.889 assertions/s.

4 tests, 4 assertions, 0 failures, 0 errors, 0 skips
```

Figure 7.3.1: *RubyCaseGen* output for Fibonacci implementation

7.4.1 Array Inference Modifications in Mirah

Mirah compiles Ruby code into Java's object model using Java's data structures. This means that Mirah automatically attempts to infer array objects as being of the Java List type. Since Kiasan is not properly developed to support Java's List types, this leads to many conflicts of supportable features. In short, it is extremely difficult to provide any useful examples to *RubyCaseGen* that use arrays without our modifications (these limitations are discussed in more detail in Section 7.5).

Because Mirah is open source, it is possible to modify the type inference implementation in order to recognize implicit arrays as being of native Java array types rather than Lists. Kiasan can handle native array types well, and this inference modification removes the conflict of features between Kiasan and Mirah, allowing us to properly test code that uses basic 1-dimensional arrays.

A few minor API changes between the use of Lists and arrays are worth noting. Firstly, Mirah by default generates immutable Lists, which means that the modification to native arrays does not actually sacrifice mutability. Secondly, Java's native arrays use the `.length` property instead of the `.size()` method, and this adjustment is made in Ruby code samples that use arrays. Note that in Ruby, *size* and *length* are both valid method names to return the size of an Array, so this API change still produces valid Ruby.

7.4.2 Minimizing Type Annotations

Although *RubyCaseGen* performs no specific behaviour on its own to allow for fewer type annotations, we discuss the implications of Mirah's type inference support, because it is relevant to the amount of annotations needed in a program.

Mirah relies heavily on its type inference in order to determine the types of variable assignments and other expressions. This means that the language itself is responsible for and able to perform type checking on variables. Typically, the only places where Mirah is unable to determine types automatically are: method parameters, return types (for complex flow paths only) and explicit casting of a polymorphic type. In these cases, Mirah uses explicit type annotation discussed earlier in this chapter, as well as a casting syntax in the form of `TheClassName(obj_to_cast)` (which is compatible with Ruby syntax and library implementations).

Because Mirah performs most of the type inference for us, including, in many cases, inference on return types, we are left with fewer types to fill in. Specifically, we can omit the use of `@local` annotations completely, as were used in *RubyEsc*. Also, in many cases, even `@return` annotations can be omitted. This leaves us with only `@param` annotations to write, which are used to translate to Mirah syntax (as discussed in Section 7.2.2).

Note that the total annotation count of the experiments is used as a metric in the case study (see Section 7.6), as one of the goals of *RubyCaseGen* is "developer ease of use".

7.4.3 Handling Successful Flow Paths

By default, *RubyCaseGen* generates all test cases that it is able to for both successful and unsuccessful program states. This feature can be turned off by passing the `--errors-only` switch to the command line tool. Although the major goal of this verification tool is to *find bugs*, it is also useful for a tool to automatically generate obvious unit tests for a given function. Kiasan provides us with many different cases, some pathological, but also some successful cases.

We therefore use these successful states to generate a special form of test with assertions on the method's return value. Figure 7.3.1 shows such a test case.

7.4.4 Handling Failure States

Catching failure states is a major goal of *RubyCaseGen*. Since we do not encode post-conditions, we instead define "failure states" as flow paths that raise an exception. Note that exceptions can be explicitly raised by assertions in a method body, and therefore post-conditions can be implemented as a set of assertions in the method.

In the case of a program exception, the generated test case does not compare the post-execution state. Instead, the tested method is called and the exception is thrown to the test suite itself. We decided against explicitly catching the exception, as this would not benefit the test code from a functional perspective, and it would require a mapping from Java exception classes back to their Ruby equivalents (the opposite mapping of the one Mirah performed).

7.4.5 Pruning Duplicate Test Cases

In some cases, Kiasan's symbolic execution engine returns multiple execution cases that all share the same initial conditions, and therefore, the exact same resulting test code. Rather than listing these duplicate cases, these test cases are automatically pruned from the generated code output.

Given two initial method states s_1, s_2, a **distinct test case** is defined formally as (p_1, p_2 are parameter names, including receiver object, of respective states, and **toRuby** refers to the function that converts $p1, p2$ into Ruby source):

$$\exists p_1, p_2 \bullet (p_1 \in s_1 \wedge p_2 \in s_2) \wedge (p_1 \notin s_2 \vee p_2 \notin s_1 \vee (p_1 = p_2 \wedge \textbf{toRuby}(p_1) \neq \textbf{toRuby}(p_2)))$$

Our duplicate check algorithm looks for all pairs of states in which the above does not hold true. Note that this not an exhaustive duplication check. For instance, parameters can contain complex objects whose own attributes and field members might be equivalent in memory, but are generated to Ruby code in different orders as follows:

```
this = Stack.new.tap do |o|
  o.elements = [1, 2]
  o.size = 2
end
```

(a) Initial state A

```
this = Stack.new.tap do |o|
  o.size = 2
  o.elements = [1, 2]
end
```

(b) Initial state B

We therefore depend on the structure Kiasan's XML format to ensure that we do not miss any duplicates. Of course, because maintaining duplicate tests does not impact test coverage in any negative way, it is okay for *RubyCaseGen* to occasionally miss some edge cases, and the heuristics in the algorithm can be improved over time.

7.5 Known Limitations

In a similar fashion to *RubyEsc*, we do not model the complete Ruby programming language in our tool. Instead we provide a working subset of the language. Unlike *RubyEsc* however, *RubyCaseGen* has different limitations. Because we translate Ruby into Mirah and pass the compiled Mirah bytecode into Kiasan, we are reliant entirely on Mirah's compatibility with Ruby's feature set and Kiasan's support for Java. We discuss the limitations of each technology below.

7.5.1 Limitations of Mirah

Since translation is performed from Ruby to Mirah, we can only support the features of Ruby that are supported by Mirah. Recall that Mirah is a fairly new language and does not support all of the functionality of the Ruby language. Mirah is also not meant to be an exact implementation of the Ruby language, and therefore there are also features that are inherently incompatible with Mirah's design. For instance, metaprogramming (and dynamic behaviours of Ruby) are not implemented in Mirah (this is not an issue, since our research does not cover Ruby metaprogramming anyhow). Nevertheless, it is still more mature than our *RubyEsc*, and supports *at least* all of the features discussed in Chapter 6.

7.5.1.1 Metaprogramming

As mentioned, Mirah supports none of Ruby's dynamic metaprogramming behaviour. Although our research also ignores the metaprogrammed behaviour of Ruby programs in general, there are some uses of metaprogramming that are extremely common and supported by our *RubyEsc* tool. Specifically, Ruby attributes are defined through the `attr_accessor` metaprogramming declaration which generates getter and setter methods for a given instance variable. Mirah supports this statement through a feature called "macros", which allows the explicit support of certain class level declarations—however Mirah does not allow explicit type declarations on these attributes, performing type inference instead. The problem is that this inference is not very reliable and causes many unwarranted compilation failures. It is therefore more difficult to rely on Mirah's use and support for idiomatic metaprogrammed attributes, and Ruby programs often need to be translated to remove the use of attributes defined with the idiomatic `attr_accessor` and `attr_reader`.

7.5.1.2 Support for Native Arrays

Mirah does not translate Ruby array objects into native Java array types, even for arrays of native types such as `int` or `double` types. Instead of native arrays, the `List` Java interface is used to manage *immutable* collections in a Mirah program. Although

this is not a problem with respect to Ruby, since Ruby has no native types, this does lead to trouble with Kiasan, as it has better support for native arrays than list types. This issue is discussed in Section 7.5.2.2. Although it is possible to explicitly specify a native Array type, all closure-based looping (using the `each` method) is performed on `List` objects only.

In the end, we decided to use a modified version of Mirah that translates implicitly defined array objects into native Java arrays rather than `List` objects. Since Mirah already uses immutable data types, there are only a few minor functional differences to our modifications, as were discussed in Section 7.4.1.

7.5.2 Limitations of Kiasan

In addition to limitations imposed by Mirah, we are also limited by the functionality of Kiasan.

7.5.2.1 Nonlinear Formulas

Kiasan uses Yices as its backend SMT solver[3] and constraint generator. This means that Kiasan's power is in turn limited by what Yices is capable of. One specific limitation of Yices is its inability to properly handle nonlinear relationships in mathematical equations. This severely limits the type of mathematical experiments we can run on the system, as we must make sure the examples are useful but also simple enough to be linear. One experiment we attempt to perform is verification of a program that performs "divide by zero"; this is a nonlinear formula. Fortunately in this case, Yices is able to detect the failure test case, but unfortunately, it also generates an incorrect "successful" test case.

[3]Kiasan currently has experimental support for Z3 as a backend theorem prover, which is expected to have better support for nonlinear relationships than Yices. However, due to technical issues, we were not able to test this experimental support.

7.5.2.2 Modeling of Java Standard Library

One other important limitation of Kiasan/Java is its lack of support for the complete standard library. Kiasan has very limited support for higher level data structures such as the `ArrayList` class, and even simpler methods like String concatenation. The lack of proper collection class modeling is particularly limiting, since Mirah translates all array usage into `List` types. Without the modifications to Mirah discussed in Section 7.4.1, we would typically receive the following error when verifying methods:

```
Ignored classes should be substituted:
  java.util.ArrayList.<init>(int) : void, thus, path abandoned.
```

7.6 Experiments Using Test Case Generation

7.6.1 Defect Detection and Test Case Generation

In order to validate the tool's ability to detect defects, we create a set of 12 Ruby programs and run these programs to observe how *RubyCaseGen* handles a range of bug classifications. Since we are not testing Ruby translation as we did in Chapter 6, we focus on higher level experiments, such as how the tool deals with **assertion errors**, **exceptions**, **null pointers**, **array bound errors**, and **arithmetic errors** (namely division by zero). We also generate some experiments to exercise the tools weaker points, such as dealing with large loops and recursion. We measure the total number of test cases generated ($CASE$), the number of pathological[4] cases detected ($FAIL$), the number of test cases that were generated incorrectly (INV), and execution time for all experiments. In total, we measured 12 files on a machine with a 3.30ghz Intel® Core™ i5-2500k CPU running Windows 7 and Mirah 0.0.11 (with custom modifications). Table 7.6.1 shows our collected data and measurements.

[4]Pathological cases are test cases that we expect to fail so we can exercise Kiasan's ability to automatically detect incorrect program logic. To do this, we craft syntactically valid programs with intentional errors. This differs from INV, which represents test cases for *correct programs* that Kiasan did not properly generate.

FILE	METH	LOC	ANN	CASE	FAIL	INV	TIME(s)
AddOne.rb	add	18	1	1	1	0	6.21
ArrayAccess.rb	element	12	1	3	2	1	6.27
Container.rb	swap	14	4	5	2	0	6.30
DispatchLookup.rb	main	19	3	1	0	0	6.64
Div0.rb	test	5	2	2	1	1	5.92
Exceptions.rb	try_div0	16	1	1	1	0	6.25
Fibonacci.rb	fib	9	2	4	0	0	6.39
Fractal.rb	run	45	2	–	–	–	6.60
IvarState.rb	main	16	3	2	1	0	6.30
NullPointer.rb	main	11	1	–	–	–	5.87
Stack.rb	use	20	1	1	0	0	6.46
Tak.rb	tak	18	4	3	0	0	7.09

Legend

METH	Method being tested	FAIL	Pathological case count
ANN	Annotation count	INV	No. of incorrect tests
CASE	Total test case count	TIME	Tool execution time

Table 7.6.1: Results of *RubyCaseGen* across various Ruby example files

We note that in some cases, specifically for the loop and recursion experiments, the tool was unable to generate any test cases (or failed due to program crash); these failures are due to limitations in Kiasan as discussed in Section 7.5.2. We note them with a dash (–) in the case count. We also discovered that it was difficult to generate examples with ample pathological cases for many bug classifications. In at least one classification (arithmetic errors), Kiasan is limited to linear formulas, which means division by zero can find a pathological case, but returns the incorrect "success" cases. We count these scenarios as invalid tests, or INV in the legend.

7.6.2 Annotation Count Comparison

7.6.2.1 Methodology

In order to verify **usability**, we compare the number of annotations needed to use *RubyCaseGen* against the number of annotations needed to use *RubyEsc*. This is important, because all Ruby programs will require extra added annotations in order to be compatible with either tool, and therefore, in order to maximize adoption rates, it is important for the tool to be as unobtrusive as possible.

Weights. It is important to note that not all annotations are created equal, and that

some annotations like `@param` and `@return` are much easier to add into a program than the more complex contract based annotations such as `@requires`, `@ensures`, `@modifies`, and `@invariant`. We therefore wish to calculate the *weighted value* of each program's annotation count, using the table below to define the weights of each annotation based on the perceived difficulty of adding each annotation.

Annotation	**Weight**
`@invariant`	4
`@ensures`	3
`@requires`	2
All Other	1

We note that the weight of specifying a pre-condition is simpler than specifying the post-condition. This is because pre-conditions do not have to reason about the logic of the method itself. Practically speaking, they are typically just simple constraints, such as $n > 0$, whereas post-conditions must capture the entire logic of the method. Invariants are even more difficult to specify, since they must capture both the pre and post states of a loop. Specifying proper loop invariants are complex enough to warrant an entire specialization of verification research [Sankaranarayanan *et al.*, 2004, Flanagan and Qadeer, 2002, Henzinger *et al.*, 2008].

Weighting is important to determine the true cost of these annotations. Because many of the annotations in *RubyCaseGen* are removed thanks to inference in Mirah, and because this can be replicated in *RubyEsc* with proper inference support, it is less important to compare the added cost of `@param` and `@return` annotations. We therefore want to compare the annotations that cannot be easily inferred by a tool.

Translation. The comparison is performed against all of the Ruby programs tested in the validation of *RubyEsc* (see Section 6.6). These programs are translated using the minimum number of annotations required to compile to a valid Mirah program and be run through *RubyCaseGen*. We remove `@ensures` clauses, because they are not used by this tool, but we maintain `@requires` clauses because they can be used to specify useful constraints when performing symbolic execution.

Test Cases Omitted. We note that we do not run our translated programs through *RubyCaseGen*. We simply prepare the programs as valid input. We do verify that

	Total Annotations		Weighted Value		
Filename	***RubyEsc***	***RubyCaseGen***	***RubyEsc***	***RubyCaseGen***	**Delta**
array.rb	1	0	1	0	-1
boolean.rb	0	0	0	0	–
class_methods.rb	1	0	1	0	-1
closures.rb	2	2	2	2	–
condition.rb	1	0	3	0	-3
contracts.rb	4	0	8	0	-8
dispatch_lookup.rb	6	2	10	2	-8
dynamic_dispatch.rb	13	1	19	1	-18
equality.rb	1	0	1	0	-1
exception.rb	7	1	13	1	-12
exception2.rb	7	0	11	0	-11
literals.rb	1	0	1	0	-1
looping.rb	1	0	1	0	-1
math.rb	7	4	12	5	-7
operators.rb	1	0	1	0	-1
stack.rb	7	2	14	3	-11

Table 7.6.2: Comparison of annotation counts needed for *RubyEsc* and *RubyCaseGen* across various equivalent Ruby example files

all programs can be parsed by our tool (which they can), but we are not interested in the number of test cases generated by the tool. This is because these modified programs are often not formulated in such a way that can generate more than one path condition (and test case). They also rely heavily on contract based assertions which are not supported by *RubyCaseGen*.

7.6.2.2 Results

Table 7.6.2 lists the results of our comparison. In most cases, the *RubyCaseGen* compatible version contains fewer annotations than the *RubyEsc* version. We observe that the weighted distribution illustrates that although *RubyCaseGen* requires annotations in some cases, these annotations are still extremely low in perceived difficulty. This contrasts with the weighted values of *RubyEsc* which are amplified by the perceived difficulty of the annotations.

It should again be noted that the most complex annotations were intentionally removed from the *RubyCaseGen* compatible programs. Although it can be argued that this skews the comparison, we believe that this distinction is the entire point of the

comparison. The fact that this tool can function without requiring complex specifications is the exact usability goal that we attempt to meet. This comparison is not meant to show which tool can perform better verification; such a comparison would be apples to oranges, as each tool performs a very distinct type of verification. Instead, this comparison aims to show which tool has a smaller adoption cost in a typical development workflow. We show that *RubyCaseGen* has a much lower development cost due to the smaller number of annotations required.

Chapter 8

Related Work

In this chapter we identify ongoing work and technologies in the realms of static verification, symbolic execution, and runtime testing that relate to our tool *RubyCorrect*, either via the technology or the methodologies and techniques used in the research. Although there are many static verification and symbolic execution tools and platforms such as *ESC/Java* [Flanagan *et al.*, 2002], *JML*4 [Chalin *et al.*, 2008], and *LOOP* [van den Berg and Jacobs, 2001], we focus on those that provide potential benefit to future research in *RubyCorrect*. In addition, although most related work is not directly relevant to dynamic languages, we identify a few that are specifically related to Ruby.

8.1 Static Verification

8.1.1 Why

Why is an intermediate verification language and static verification (VCGen) tool [Filliâtre and Marché, 2007]. We mention this research because it is another IVL supporting many of the features of both Boogie and Pilar, but we did not consider it in the comparison, mainly due to the need to limit the scope (in time and space) of the thesis effort and manuscript. Its tooling is built for Java and C VCGen, with back-end support for many theorem provers including *Isabelle*, for which there is currently

experimental support in Boogie [Böhme *et al.*, 2010] but no support in Pilar. *Why* is published under GPLv2, making it open source and easily modifiable.

8.1.2 Diamondback Ruby

Diamondback Ruby (DRuby) is an active research project by the University of Maryland in order to attempt to provide type inference to the Ruby language [An *et al.*, 2011]. DRuby is interesting because it provides an annotation syntax that can specify the behaviour of Ruby's complete object model, and is more complete than YARD's annotation syntax. DRuby also has a fairly complete mapping of annotations for methods in Ruby's standard library, which means it has practical usage for real world code. Applying this more complex annotation syntax could yield better results for *RubyCorrect* in the future.

8.1.3 Laser

Laser is a static analysis and lint-like analysis tool for the Ruby programming language [Edgar, 2011]. It translates Ruby programs into an intermediate representation and performs a best-effort type inference based on work by DRuby (above) and Ecstatic [Madsen *et al.*, 2007]. This IR is then used to provide basic analyses such as incorrect parameter counts on method calls and closures, missing method errors, uncaught exceptions, as well as dead code and unused variable checking. The tool is written in Ruby and is available as open source under the AGPLv3 license. The methodology described in this research for performing type inference on Ruby programs could eventually be applied to *RubyCorrect* in order to reduce the number of annotations required to perform verifications.

8.2 Symbolic Execution

8.2.1 Java Path Finder

Java Path Finder (JPF) is a suite of verification tools, including symbolic execution, for JVM bytecode [Păsăreanu *et al.*, 2008]. JPF is developed by NASA, is written in Java and is open source under NASA's NOSA1.3 license. We focus mainly on the symbolic execution portion of this tool because it is extremely similar in functionality to Kiasan, as well as *RubyCaseGen*. Specifically, JPF can also perform test case generation directly to executable code, except it focuses only on Java code and uses JUnit. However, since *RubyCaseGen* performs translation down to Java, it would be feasible to perform future experiments with JPF as an alternative to Kiasan as the backend symbolic execution engine.

8.2.2 Valigator

Valigator is another symbolic execution engine for the JVM (though written in Scala) [Henzinger *et al.*, 2008]. This research is specifically interesting because it can perform inference on loop invariants and bounds, which could be used to reduce the most complex of the annotations required by *RubyCorrect*. However, the tool takes input in a custom C-like language, and therefore would be more difficult to integrate into our architecture. Valigator is also not available under an open source license and is only available as a binary, but the source is available under academic-only licenses on request.

8.3 Runtime and Fuzz Testing

8.3.1 Heckle

Heckle is a runtime testing framework for the Ruby language [Clark and Davis, 2012]. It performs **fuzz testing**, which introduces random data into program inputs to detect program faults [Godefroid *et al.*, 2008]. It is written in Ruby and available under

the MIT open source license. Heckle works specifically by generating an AST of a given Ruby program and performing controlled mutations on certain nodes to, for instance, replace literals with random values, and swap if-then-else condition bodies. We mention this work because it is similar to the symbolic-to-concrete value resolution done by a SymExec tool like Kiasan. However, unlike Kiasan, Heckle requires pre-existing tests with high path coverage to validate whether a change detected a program fault, whereas SymExec can generate those test cases from program execution alone. One benefit of a tool like Heckle is that because it executes at runtime, it inherently supports all of the functionality of Ruby, even its most dynamic behaviours. It could therefore be interesting to see a combination of Heckle and *RubyCorrect* could improve coverage of the Ruby language while still maintaining its practical benefits.

Chapter 9

Conclusion and Future Work

Throughout this thesis we discussed the methodologies and techniques to performing static verification of Ruby programs using extended static checking and symbolic execution through two novel tools, *RubyEsc* and *RubyCaseGen*, respectively. In this chapter, we discuss our findings and our vision for improving the *RubyCorrect* toolchain in the future.

9.1 Extended Static Checking with Ruby

The initial hypothesis in attempting ESC for a dynamic language like Ruby was that even a dynamic language should have some identifiable constant properties. Chapter 3 results are in support of this hypothesis. The experiment conducted to test this hypothesis involved building the *RubyEsc* component of the *RubyCorrect* toolchain and using this tool to observe whether a Ruby program could in fact be translated and verified as a static program.

Overall, the results from *RubyEsc* were positive. We were able to verify simple-yet-idiomatic programs on a small scale without too many technical issues. We were able to verify programs using Ruby's basic object model, dynamic dispatch, exception handling, and closures. Translating the syntax from Ruby to Boogie proved to not be overly difficult, although there are some Ruby features such as local or parameter

variable containers supporting multiple orthogonal types that are not yet covered by our tool and would require more work to properly model in Boogie.

One issue with our tests was that we were heavily limited by our minimal model of the full Ruby standard library. Modeling the full standard library would be very time consuming, as Ruby's stdlib is quite large. The core library (of Ruby 1.9.3) alone contains 17 modules, 87 classes, and 1603 methods—each method would require contracts, which are nontrivial to write—and that is just the core library. The rest of the standard library (of which libraries like *thread* are a part) is much larger. (To give a measure of the magniture of this effort, the JML specifications of some of the standard Java libraries took several person-years to develop.)

Speaking more generally, we also found that writing good specifications for Ruby programs is difficult, though this difficulty is not at all limited to Ruby. ESC in general is no easy task, and it is certainly debatable how much effort the Ruby community would be willing to devote to writing these relatively costly specifications when runtime testing is so much cheaper. Nevertheless, we have shown that with good specifications, our tool can provide a reliable and effective means of verifying program logic. In other words, we have shown that, for a subset of Ruby, it is possible to have statically verifiable programs in a dynamic language with extremely minor program modifications.

9.2 Symbolic Execution with Ruby

One of the appeals of symbolic execution is that it allows us to inspect program state before and after the execution for a wide range of initial conditions. In other words, symbolic execution gives us the ability to determine inputs that will cause a program to pass or fail. This is extremely powerful, especially in the Ruby community, which relies heavily on test-driven development (TDD). The ability to generate test cases is the feature of *RubyCaseGen* that provides real potential impact for the Ruby community, as there is currently no other tool that can do this.

Our initial results with *RubyCaseGen* are minimal, but promising. We have shown that there are some classifications of programs that could be run through our tool

to create useful test cases which could theoretically improve QA and productivity, though a larger case study (as future work) would be needed to confirm this. We have also shown that using this tool is significantly easier than using a full static checker like *RubyEsc* (see Section 7.6.2). This is important, because it is a key goal of the tool. We were not able to adequately check whether the tool could be used to detect "hidden" bugs, but this would require our tool to be able to handle large production-ready codebases, which we cannot yet do.

9.3 Future Work

Improving Coverage. The most impactful improvement that could be made to *RubyCorrect* would be to increase the support for various features of Ruby. We acknowledged from the start that we omit support for some very basic functionalities of Ruby, but we want to improve this support. Specifically, we would like to see a more complete model of Ruby's standard library for *RubyEsc* to use in its verification, and we would like to see proper handling of arrays and standard libraries in *RubyCaseGen*, although the latter would require moving away from Java as the underlying target language.

Case Studies. Once the tools are at a point where they can process full real world programs, it would be possible to perform a practical case study of using extended static checking to leverage design-by-contract programming in Ruby, as well as how symbolic execution can be used to find defects and generate test cases for existing software systems. This would be the ultimate test of *RubyCorrect*'s practicality.

Other Experiments. Of course, there are different avenues that can be taken to reach this ultimate goal. Some of these new avenues were briefly addressed in Chapter 8. Specifically, tools like *Java Path Finder*, *Valigator* can be experimented with to potentially improve the coverage of `ruby_case_gen` (as mentioned, we had difficulties testing code generated by Mirah through Kiasan, perhaps another symbolic execution engine would perform better). We can also look at improving type inference and type specifications in general by following the lead of projects like *DRuby* and *Laser*. Finally, there are some potential lessons to be learned from runtime testing

that can be merged into *RubyCorrect* for a hybrid approach to static verification—symbolic execution mostly tries to do this, but perhaps we can leverage more of the very versatile Ruby runtime.

9.4 Conclusion

We have seen that Ruby can in fact be translated into a multitude of static forms given a very minimal set of program annotations. The translation to Mirah is most striking—we have shown that a program written entirely in Ruby syntax can be translated to a purely static Java program with only a handful of annotations. Furthermore, we show through our novel tool, *RubyCorrect*, that there are practical benefits to performing design-by-contract as well as the potential QA and productivity benefits in generating test cases for Ruby programs. In essence, our tool, along with our method of annotating Ruby programs, shows that static analysis is a viable avenue of research for a dynamic language such as Ruby. Rather than focusing on improving type inference as a basis to prove that Ruby can be verified, we use annotations as training wheels to show that annotated Ruby can be verified, and those verifications can prove useful. We leave room for research in dynamic language type inference to fill in the gaps and remove the requirement for some of the extra type annotations. We believe that once this research moves forward, it will eventually be possible, using the techniques outlined in this thesis, to perform static analysis on completely unmodified Ruby programs.

Bibliography

[An *et al.*, 2011] Jong-hoon David An, Avik Chaudhuri, Jeffrey S. Foster, and Michael Hicks. Dynamic inference of static types for Ruby. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, Austin, TX, USA, January 2011.

[Anderson and Drossopoulou, 2006] C. Anderson and S. Drossopoulou. *Type inference for Javascript.* PhD thesis, Department of Computing, Imperial College London, March 2006.

[Aycock, 2000] J. Aycock. Aggressive type inference. In *Proceedings of the 8th International Python Conference*, pages 11–20, 2000.

[Barnes, 2003] John Barnes. *High Integrity Software – the SPARK Approach to Safety and Security.* Addison-Wesley, 2003.

[Barnett and Leino, 2005] Michael Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In Michael D. Ernst and Thomas P. Jensen, editors, *PASTE*, pages 82–87. ACM, 2005.

[Barnett *et al.*, 2005] Michael Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In Bertrand Meyer and Jim Woodcock, editors, *Proceedings of the International Workshop on the Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS 2004)*, volume 4171 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2005.

[Barrett and Tinelli, 2007] Clark Barrett and Cesare Tinelli. CVC3. In Damm and Hermanns [2007], pages 298–302.

[Belt *et al.*, 2011] Jason Belt, John Hatcliff, Robby, Patrice Chalin, David Hardin, and Xianghua Deng. Bakar kiasan: Flexible contract checking for critical systems using symbolic execution. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 58–72. Springer, 2011.

[Böhme *et al.*, 2010] Sascha Böhme, Michal Moskal, Wolfram Schulte, and Burkhart Wolff. HOL-Boogie - an interactive prover-backend for the verifying C compiler. *J. Autom. Reasoning*, 44(1-2):111–144, 2010.

[Cartwright and Fagan, 2004] Robert Cartwright and Mike Fagan. Soft typing. *SIGPLAN Not.*, 39:412–428, April 2004.

[Cartwright and Felleisen, 1996] Robert Cartwright and Matthias Felleisen. Program verification through soft typing. *ACM Computing Surveys*, 28:349–351, June 1996.

[Chalin *et al.*, 2008] P. Chalin, P. James, and G. Karabotsos. JML4: Towards an industrial grade IVE for Java and next generation research platform for JML. *Verified Software: Theories, Tools, Experiments*, pages 70–83, 2008.

[Chrząszcz *et al.*, 2009] J. Chrząszcz, M. Huisman, and A. Schubert. BML and Related Tools. In *Formal Methods for Components and Objects*, pages 278–297. Springer, 2009.

[Clark and Davis, 2012] Kevin Clark and Ryan Davis. Heckle. `http://ruby.sadi.st/Heckle.html` (accessed July 30 2012), 2012.

[Cohen *et al.*, 2009] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. Vcc: A practical system for verifying concurrent c. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42. Springer Berlin / Heidelberg, 2009. 10.1007/978-3-642-03359-9_2.

[Damm and Hermanns, 2007] Werner Damm and Holger Hermanns, editors. *Proceedings of teh 19th International Conference on Computer Aided Verification (CAV 2007)*, volume 4590 of *Lecture Notes in Computer Science*. Springer, 2007.

[Deng *et al.*, 2006] Xianghua Deng, Robby, and John Hatcliff. Kiasan: A verification and test-case generation framework for Java based on symbolic execution. In *ISoLA*, page 137. IEEE, 2006.

[Deng *et al.*, 2007a] Xianghua Deng, Robby, and J. Hatcliff. Kiasan/kunit: Automatic test case generation and analysis feedback for open object-oriented systems. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007. TAICPART-MUTATION 2007*, pages 3 –12, sept. 2007.

[Deng *et al.*, 2007b] Xianghua Deng, Robby, and John Hatcliff. Towards a case-optimal symbolic execution algorithm for analyzing strong properties of object-oriented programs. *Software Engineering and Formal Methods, IEEE International Conference on*, 0:273–282, 2007.

[Deng *et al.*, 2012] Xianghua Deng, Jooyong Lee, and Robby. Efficient and formal generalized symbolic execution. *Autom. Softw. Eng.*, 19(3):233–301, 2012.

[Dijkstra, 1975] E.W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[Dolby, 2005] J. Dolby. Using static analysis for IDE's for dynamic languages. In *The Eclipse Languages Symposium*, 2005.

[Dutertre and de Moura, 2006] B. Dutertre and L. de Moura. The Yices SMT solver. Tool paper at `http://yices.csl.sri.com/tool-paper.pdf`, August 2006.

[ECMA, 2009] ECMA. Ecma-262: Ecmascript language specification, 5th edition. `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf`, December 2009.

[Edgar, 2011] Michael Joseph Edgar. Static analysis for Ruby in the presence of gradual typing. Technical Report TR2011-686, Dartmouth College, 2011.

[Filliâtre and Marché, 2007] Jean-Christophe Filliâtre and Claude Marché. The why/krakatoa/caduceus platform for deductive program verification. In Damm and Hermanns [2007], pages 173–177.

[Flanagan and Qadeer, 2002] Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. *SIGPLAN Not.*, 37(1):191–202, January 2002.

[Flanagan *et al.*, 2002] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. *SIGPLAN Not.*, 37(5):234–245, May 2002.

[Flanagan, 2005] D. Flanagan. *Java in a Nutshell*. O'Reilly Media, 2005.

[Frank, 2005] B. A. Frank. The Fantom programming language. `http://www.fantom.org/`, 2005.

[Furr *et al.*, 2009a] Michael Furr, Jong-hoon (David) An, and Jeffrey S. Foster. Profile-guided static typing for dynamic scripting languages. *SIGPLAN Not.*, 44:283–300, October 2009.

[Furr *et al.*, 2009b] Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. Static type inference for Ruby. In *Proceedings of the 2009 ACM symposium on Applied Computing*, SAC '09, pages 1859–1866, New York, NY, USA, 2009. ACM.

[Gal *et al.*, 2009] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 465–478, New York, NY, USA, 2009. ACM.

[Godefroid *et al.*, 2008] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium*. The Internet Society, 2008.

[Grigore, 2007] Radu Grigore. Efficiency of Extended Static Checkers. Technical report, PhD Research Plan. UCD Dublin, December 2007.

[Grigore, 2009] Radu Grigore. FreeBoogie. `http://code.google.com/p/freeboogie`, July 2009.

[Havelund and Pressburger, 2000] K. Havelund and T. Pressburger. Model checking Java programs using Java pathfinder. *International Journal on Software Tools for*

Technology Transfer (STTT), 2(4):366–381, 2000.

[Henry, 2006] Kevin Henry. A crash overview of Groovy. *ACM Crossroads*, 12(3):5, 2006.

[Henzinger *et al.*, 2008] Thomas Henzinger, Thibaud Hottelier, and Laura KovÃącs. Valigator: A verification tool with bound and invariant generation. In Iliano Cervesato, Helmut Veith, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 5330 of *Lecture Notes in Computer Science*, pages 333–342. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-89439-1_24.

[Hickey, 2008] Rich Hickey. The Clojure programming language. In Johan Brichau, editor, *DLS*, page 1. ACM, 2008.

[Holkner and Harland, 2009] Alex Holkner and James Harland. Evaluating the dynamic behaviour of python applications. In *Proceedings of the Thirty-Second Australasian Conference on Computer Science - Volume 91*, ACSC '09, pages 19–28, Darlinghurst, Australia, Australia, 2009. Australian Computer Society, Inc.

[Huang *et al.*, 2004] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th international conference on World Wide Web*, WWW '04, pages 40–52, New York, NY, USA, 2004. ACM.

[Huisman and Jacobs, 2000] M. Huisman and B. Jacobs. Java program verification via a Hoare logic with abrupt termination. *Fundamental Approaches to Software Engineering*, pages 284–303, 2000.

[Hunt *et al.*, 2006] J.J. Hunt, F.B. Siebert, P.H. Schmitt, and I. Tonin. Provably correct loops bounds for realtime java programs. *JTRES*, 6:162–169, 2006.

[James *et al.*, 2008] P.R. James, P. Chalin, L. Giannas, and G. Karabotsos. Distributed, multi-threaded verification of Java programs. In *Seventh International Workshop on Specification and Verification of Component-Based Systems*, page 3. SAVCBS, 2008.

[King, 1976] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19:385–394, July 1976.

[Kramer, 1999] Douglas Kramer. API documentation from source code comments: a case study of Javadoc. In *Proceedings of the 17th annual international conference on Computer documentation*, SIGDOC '99, pages 147–153, New York, NY, USA, 1999. ACM.

[Kramer, 2001] D. Kramer. How to write doc comments for Javadoc. *Javadoc Home Page: http://jsp2. java. sun/products/jdk/javadoc/writingdoccomments/index. html*, 2001.

[Leavens *et al.*, 1999] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999.

[Leavens *et al.*, 2006] G.T. Leavens, A.L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.

[Leavens *et al.*, 2011] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David R. Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, and Daniel M. Zimmerman. JML Reference Manual. Available from `http://www.jmlspecs.org`, May 2011.

[Leino, 1998] K. Rustan M. Leino. Extended static checking. In David Gries and Willem P. de Roever, editors, *PROCOMET*, volume 125 of *IFIP Conference Proceedings*, pages 1–2. Chapman & Hall, 1998.

[Leino, 2008] K. Rustan M. Leino. This is Boogie 2. Technical Report KRML 178, Microsoft Research, June 2008. Available from `http://research.microsoft.com/en-us/um/people/leino/papers/krml178.pdf`.

[Leino, 2010a] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010.

[Leino, 2010b] K. Rustan M. Leino. Verifying concurrent programs with Chalice. In Gilles Barthe and Manuel V. Hermenegildo, editors, *VMCAI*, volume 5944 of *Lecture Notes in Computer Science*, page 2. Springer, 2010.

[Liang, 1999] S. Liang. *The Java native interface: programmer's guide and specification*. Addison-Wesley Professional, 1999.

[Madsen *et al.*, 2007] M. Madsen, P. Sørensen, and K. Kristensen. Ecstatic-type interference for Ruby using the cartesian product algorithm. *Master's thesis, Aalborg University*, 2007.

[Marvie, 2008] R. Marvie. An introduction to test-driven code generation. *The Python Papers*, 2(4), 2008.

[Meunier, 1995] R. Meunier. The pipes and filters architecture. In *Pattern languages of program design*, pages 427–440. ACM Press/Addison-Wesley Publishing Co., 1995.

[Microsoft Research, 2012] Microsoft Research. The HAVOC property checker. `http://research.microsoft.com/en-us/projects/havoc/`, 2012.

[Mount *et al.*, 2004] S. N. I. Mount, R. M. Newman, R. J. Low, and A. Mycroft. Exstatic: a generic static checker applied to documentation systems. In *Proceedings of the 22nd annual international conference on Design of communication: The engineering of quality documentation*, SIGDOC '04, pages 52–57, New York, NY, USA, 2004. ACM.

[Odersky *et al.*, 2008] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, USA, 1st edition, 2008.

[Odersky, 2007] M. Odersky. The Scala language specification, version 2.7. *Programming Methods Laboratory, EPFL*, 2007.

[Păsăreanu *et al.*, 2008] Corina S. Păsăreanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape. Combining unit-level symbolic execution and system-level concrete execution for testing nasa

software. In *Proceedings of the 2008 international symposium on Software testing and analysis*, ISSTA '08, pages 15–26, New York, NY, USA, 2008. ACM.

[Ramakrishnan and Rehof, 2008] C. R. Ramakrishnan and Jakob Rehof, editors. *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*, volume 4963 of *Lecture Notes in Computer Science*. Springer, 2008.

[Richards *et al.*, 2010] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of javascript programs. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 1–12, New York, NY, USA, 2010. ACM.

[Robby *et al.*, 2003] Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: An extensible and highly-modular model checking framework. In *Proceedings of the 9th European Software Engineering Conference held jointly with the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 267–276, 2003.

[Robby, 2007] Robby. *Sireum: A Software Analysis Platform*. SAnToS, Kansas State Univerity, February 2007. Available from `http://sireum.org`.

[Ruby on Rails, 2010] Ruby on Rails. Ruby on Rails. `http://www.rubyonrails.org` (accessed November 25 2010), 2010.

[Ruby, 2010] Ruby. About Ruby. `http://ruby-lang.org/en/about`, 2010.

[Saeed, 2008] M. Saeed & F. Saeed. Systematic review of verification & validation in dynamic languages. Master's thesis, Blekinge Institute of Technology, Sweden, August 2008.

[Salib, 2004] M. Salib. *Starkiller: A static type inferencer and compiler for Python*. PhD thesis, Massachusetts Institute of Technology, May 2004.

[Sankaranarayanan *et al.*, 2004] Sriram Sankaranarayanan, Henny Sipma, and Zohar Manna. Non-linear loop invariant generation using Gröbner bases. In *POPL*, pages 318–329. ACM, 2004.

[Saxena *et al.*, 2010] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for javascript. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 513–528, Washington, DC, USA, 2010. IEEE Computer Society.

[Segal and Chalin, 2012] Loren Segal and Patrice Chalin. A comparison of intermediate verification languages: Boogie and Sireum/Pilar. In *Proceedings of the 4th International Conference on Verified Software: Theories, Tools, Experiments*, pages 130–145, Philadelphia, PA, USA, 2012.

[Segal, 2012] Loren Segal. YARD: Yay! A Ruby Documentation Tool. `http://yardoc.org` (accessed July 30 2012), 2012.

[Tateishi *et al.*, 2006] Takaaki Tateishi, Hisashi Miyashita, Kouichi Ono, and Shin Saito. Automated verification tool for DHTML. *Automated Software Engineering, International Conference on*, 0:363–364, 2006.

[Tschannen *et al.*, 2011] Julian Tschannen, Carlo A. Furia, Martin Nordio, and Bertrand Meyer. Verifying Eiffel programs with Boogie. *CoRR*, abs/1106.4700, 2011.

[van den Berg and Jacobs, 2001] Joachim van den Berg and Bart Jacobs. The LOOP compiler for Java and JML. In *TACAS*, pages 299–312, 2001.

[van Heesch, 2011] Dimitri van Heesch. Doxygen. `http://doxygen.org`, 2011.

[Working With Rails, 2010] Working With Rails. High Profile Organisations using Rails. `http://www.workingwithrails.com/high-profile-organisations` (accessed November 25 2010), 2010.

[Xie and Aiken, 2006] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *15th USENIX Security Symposium*, pages 179–192, July 2006.

Appendix A

Modeling Ruby Arrays in Boogie

```
# @local [Array<Fixnum>] arr
def main
  arr = [1,2,3]
  assert arr[1] == 2
end
```

(a) Ruby

```
procedure #main$$ROOTCLASS(self: VALUE)
    returns ($result: VALUE, $exception: VALUE)
    modifies $heap; {
  var arr: VALUE;
  var ARRAY$1: VALUE;
  var retval$1: VALUE;
  var retval$2: VALUE;
  var retval$3: VALUE;
  var retval$4: VALUE;
  var retval$5: VALUE;
  var retval$6: VALUE;
  $exception := $nil;

  // arr = [1,2,3]:
  call retval$1, $exception := Array#initialize$Array(ARRAY$1);
  call retval$2, $exception := Array#push$Array$Fixnum(ARRAY$1, 1);
  call retval$3, $exception := Array#push$Array$Fixnum(ARRAY$1, 2);
  call retval$4, $exception := Array#push$Array$Fixnum(ARRAY$1, 3);
  arr := ARRAY$1;

  // assert arr[1] == 2:
  call retval$5, $exception := Array#$aref$Array$Fixnum(arr, 1);
  call retval$6, $exception := Object#$eq$Object$Fixnum(retval$5, 2)
     ;
  assert retval$6 == $true;

rescueBlock:
}
```

(b) Boogie

Figure A.0.1: Translating array initialization and access to Boogie

Appendix B

Translating Closures from Ruby to Boogie

```ruby
# @param [Fixnum] n
def add(n)
  yield(n + 3)
end

def main
  x = 1
  # @ensures x == old(x) + y
  add(2) do |y|
    x += y
  end
  assert x == 6
end
```

(a) Ruby

```
procedure #add$$ROOT$Fixnum(self: VALUE, n: VALUE)
    returns ($result: VALUE, $exception: VALUE)
    modifies $heap;
{
  $exception := $nil;
rescueBlock:
}

procedure {:inline 100} #add$$ROOT$Fixnum$lambda$#main$$ROOT$1(
  self: VALUE, n: VALUE, self$lambda$in: VALUE, x$lambda$in: VALUE)
    returns ($result: VALUE, $exception: VALUE, x$lambda$out: VALUE)
    modifies $heap;
{
  var y$lambda$out: VALUE;
  var retval$1: VALUE;
  var retval$2: VALUE;
  $exception := $nil;
  x$lambda$out := x$lambda$in;
  call retval$1, $exception := Fixnum#$add$Fixnum$Fixnum(n, 3);
  y$lambda$out := retval$1;
  call retval$2, $exception :=
    Fixnum#$add$Fixnum$Object(x$lambda$out, y$lambda$out);
  x$lambda$out := retval$2;
  assert x$lambda$out == x$lambda$in + y$lambda$out;
rescueBlock:
}

procedure #main$$ROOT(self: VALUE)
    returns ($result: VALUE, $exception: VALUE)
    modifies $heap;
{
  var x: VALUE;
  var retval$1: VALUE;
  var retval$2: VALUE;
  $exception := $nil;
  x := 1;
  call retval$1, $exception, x :=
    #add$$ROOT$Fixnum$lambda$#main$$ROOT$1(self, 2, self, x);
  call retval$2, $exception := Object#$eq$Fixnum$Fixnum(x, 6);
  assert retval$2 == $true;
rescueBlock:
}
```

(b) Boogie

Appendix C

RubyEsc Preamble

preamble.rb

```
# @core
class Object
  # @return [self]
  # @ensures "$result != $nil"
  def self.new
    object = allocate
    object.initialize
    return object
  end

  # @return [self]
  # @ensures "$result != $nil"
  def self.allocate; end

  def initialize; end

  # @param [Object] other
  # @ensures "self == other ==> $result == $true"
  # @ensures "self != other ==> $result == $false"
  # @return [Boolean]
  # @pure
  def ==(other) end

  # @param [Object] other
  # @ensures "self != other ==> $result == $true"
  # @ensures "self == other ==> $result == $false"
  # @return [Boolean]
```

```
  # @pure
  def !=(other) end

  # @ensures "self == $nil ==> $result == $true"
  # @return [Boolean]
  # @pure
  def nil?; end
end

# @core
class Fixnum < Object
  # @param [Fixnum] other
  # @ensures "$result == self + other"
  # @return [Fixnum]
  # @pure
  def +(other) end

  # @param [Fixnum] other
  # @ensures "$result == self - other"
  # @return [Fixnum]
  # @pure
  def -(other) end

  # @param [Fixnum] other
  # @ensures "$result == self * other"
  # @return [Fixnum]
  # @pure
  def *(other) end

  # @param [Fixnum] other
  # @ensures "$result == self / other"
  # @return [Fixnum]
  # @pure
  def /(other) end

  # @param [Fixnum] other
  # @ensures "$result == self % other"
  # @return [Fixnum]
  # @pure
  def %(other) end

  # @param [Fixnum] other
  # @ensures "self < other ==> $result == $true"
  # @return [Boolean]
  # @pure
  def <(other) end
```

```
# @param [Fixnum] other
# @ensures "self > other ==> $result == $true"
# @return [Boolean]
# @pure
def >(other) end

# @param [Fixnum] other
# @ensures "self <= other ==> $result == $true"
# @return [Boolean]
# @pure
def <=(other) end

# @param [Fixnum] other
# @ensures "self >= other ==> $result == $true"
# @return [Boolean]
# @pure
def >=(other) end

# @ensures "$result == -self"
# @return [Fixnum]
# @pure
def -@; end

# @ensures $result == -self if self < 0
# @ensures $result == self if self >= 0
# @return [Fixnum]
# @pure
def abs; end

# @param [Fixnum] n
# @requires n >= self
# @return [Fixnum]
def upto(n)
  i = self
  # @invariant (i - old(i)).abs <= 1
  while i <= n
    yield(i)
    i += 1
  end
  self
end

# @param [Fixnum] n
# @requires n <= self
# @return [Fixnum]
def downto(n)
  i = self
```

```
    # @invariant (i - old(i)).abs <= 1
    while i >= n
      yield(i)
      i -= 1
    end
    self
  end
end

# @core
class Boolean
  # @ensures "self == $true ==> $result == $false"
  # @ensures "self == $false ==> $result == $true"
  # @return [Boolean]
  # @pure
  def !; end
end

# @core
class TrueClass < Boolean
end

# @core
class FalseClass < Boolean
end

# @core
class Exception < Object; end

# @core
class RuntimeException < Exception; end

# @core
class NilClass
  # @ensures "$result == $true"
  # @pure
  def !; end
end

# @ivar [Array<$T>] elements
# @ivar [Fixnum] size
# @core
class Array < Object
  # @ensures "(forall x:VALUE :: $arrget($heap[self][Array$elements
     ], x) == $nil)"
  # @ensures @size == 0
  # @modifies @size
```

```
# @modifies @elements
def initialize; end

# @param [$T] element
# @ensures "$arrget($heap[self][Array$elements], old($heap[self][
    Array$size])) == element"
# @ensures "(forall x:VALUE :: x != old($heap[self][Array$size])
    ==> $arrget($heap[self][Array$elements], x) == old($arrget(
    $heap[self][Array$elements], x)))"
# @ensures @size == old(@size) + 1
# @modifies @elements
# @modifies @size
def push(element) end

# @ensures @size == old(@size) - 1
# @ensures "$result == old($arrget($heap[self][Array$elements],
    $heap[self][Array$size] - 1))"
# @ensures "$arrget($heap[self][Array$elements], $heap[self][
    Array$size] - 1) == $nil"
# @ensures "(forall x:VALUE :: x != $heap[self][Array$size] ==>
    $arrget($heap[self][Array$elements], x) == old($arrget($heap[
    self][Array$elements], x)))"
# @modifies @elements
# @modifies @size
# @return [$T]
def pop; end

# @param [Fixnum] idx
# @ensures "$result == $arrget($heap[self][Array$elements], idx)"
# @ensures $result == nil if idx >= @size
# @return [$T]
# @pure
def [](idx) end

# @param [Fixnum] idx
# @param [$T] value
# @ensures "$arrget($heap[self][Array$elements], idx) == value"
# @ensures $result == value
# @ensures "(forall x:VALUE :: x != idx ==> $arrget($heap[self][
    Array$elements], x) == old($arrget($heap[self][Array$elements
    ], x)))"
# @ensures @size == idx+1 if idx+1 > @size
# @modifies @elements
def []=(idx, value) end

# @ensures $result == @size
# @return [Fixnum]
```

```
  # @pure
  def size; end

  # @local [Fixnum] i
  def each
    i = 0
    # @invariant i >= 0
    while i < @size
      yield(@elements[i])
      i += 1
    end
  end

  # @local [Fixnum] i
  def each_with_index
    i = 0
    # @invariant i >= 0
    while i < @size
      yield(@elements[i], i)
      i += 1
    end
  end
end
```

Appendix D

RubyEsc Example Programs

array.rb

```
# @local [Array<Fixnum>] x
def array_test
  x = []
  assert x.size == 0
  assert x[0] == nil
  x.push 1
  assert x.size == 1
  assert x[0] == 1
  assert x[1] == nil
  x.pop
  assert x.size == 0
end

def array_test2
  x = []
  assert x.size == 0
  x.push 1
  assert x[0] == 1
  assert x.size == 1
  x.push 2
  assert x[0] == 1
  assert x.size == 2
  x.push 3
  assert x[0] == 1
  assert x[1] == 2
  assert x[2] == 3
  assert x.size == 3
```

```
  assert x.pop == 3
  assert x.size == 2
  assert x[2].nil?
  assert x[1] == 2
  assert x[0] == 1
  assert x.pop == 2
  assert x.pop == 1
  assert x.pop == nil
  assert x.pop == nil
  assert x.pop == nil
  assert x.pop == nil
  assert x.pop == nil
  assert x.size == 0
  assert x.size == -3
end

def array_test3
 x = [1,2,3]
 assert x[0] == 1
 assert x[1] == 2
 assert x[2] == 3
end
```

boolean.rb

```
def test_truthy
  truthy = false
  assert truthy
end

def test_truthy2
  truthy2 = true
  assert truthy2
end
```

class_methods.rb

```
class A
  # @return [A]
  def self.creator
    return new
  end
end

def main
```

```
  A.creator
end
```

closures.rb

```
# @param [Fixnum] n
def add(n)
  yield(n + 3)
end

def main
  x = 1
  # @ensures x == old(x) + y
  add(2) do |y|
    x += y
  end
  assert x == 6
end
```

condition.rb

```
# @ensures $result == 9
def if_then
  x = 10
  if x == 2
    x = 5
  else
    x = 9
  end
  return x
end
```

contracts.rb

```
class Math
  # @return [Fixnum]
  # @ensures $result == 100
  def one_hundred; return 100 end

  # @return [Fixnum]
  # @ensures $result == one_hundred + 100
  def two_hundred; return 200 end
end
```

dispatch_lookup.rb

```
class A
  # @param [Fixnum] n
  # @return [Fixnum]
  # @ensures $result == n + 5
  def five(n)
    return n + 5
  end
end

class B < A
  # @ensures $result == 10
  # @return [Fixnum]
  def run
    return self.five(5)
  end
end

class C < B; end

class D
  # @param [C] c
  def run_all(c)
    assert c.run == 10
  end
end

def main
  D.new.run_all(C.new)
end
```

dynamic_dispatch.rb

```
class Base
  # @return [Fixnum]
  # @ensures $result == 5
  # @pure
  def foo
    return 5
  end
end

class A < Base
  # @return [Fixnum]
  # @ensures $result == 10
  # @pure
```

```
  def foo
    return 10
  end
end

class B < A
  # @param [Base] object
  # @return [Fixnum]
  # @ensures $result == object.foo
  # @pure
  def bar(object)
    return object.foo
  end
end

# @local [B] b
# @local [A] a
# @local [Base] base
def main
  b = B.new
  a = A.new
  base = Base.new
  assert b.bar(a) == 10
  assert b.bar(base) == 5
end
```

equality.rb

```
def equality_test; assert !false end
def equality_test2; assert !true end

# @local [Boolean] x
def equality_test3
  x = !true
  assert x
end

def equality_test4
  assert true == !true
end
```

exception.rb

```
class DivideByZeroException < Exception
end
```

```ruby
class Div0
  # @param [Fixnum] n
  # @return [Fixnum]
  # @ensures $exception != nil if n == 0
  # @ensures $result == 10 / n if n != 0
  # @raise [DivideByZeroException]
  def ten_div_by(n)
    raise DivideByZeroException if n == 0
    return 10 / n
  end
end

class Main
  # @return [Fixnum]
  # @local [Div0] div
  # @ensures $result == 0
  def try_div0
    div = Div0.new
    div.ten_div_by(0)
    return 1
  rescue
    return 0
  end
end
```

exception2.rb

```ruby
class A
  # @raise [Exception]
  def foo
    raise Exception
  end
end

# @ivar [Fixnum] counter
class B
  # @local [A] a
  # @modifies @counter
  def main
    @counter = 0
    call1
    assert @counter == 2
  end

  # @ensures @counter == old(@counter) + 2
```

```
  # @modifies @counter
  def call1
    call2
    @counter += 1
  end

  # @ensures @counter == old(@counter) + 1
  # @modifies @counter
  def call2
    @counter += 1
  end
end
```

literals.rb

```
# @ivar [String] otherval2
class Stack
  def initialize
    @otherval = "HELLO"
    @arr = []
    @arr_size = 0
  end
end
```

looping.rb

```
def loop
  j = 9
  i = 0
  # @invariant j + i == 9
  while i < 9
    j = 9 - i
  end
end

# @local [Fixnum] counter
def loop2
  counter = 0
  # @local [Fixnum] x
  [1,2,3].each {|x| counter += x }
  assert counter == 6
end

def loop3
  counter = 0
```

```
  # @local [Fixnum] v
   [1,2,3].each_with_index {|x,v| counter += v }
  assert counter == 3
end
```

math.rb

```
# @requires x >= 0
# @ensures $result == x + y - 2
# @param [Fixnum] x
# @param [Fixnum] y
# @return [Fixnum]
def inc(x, y)
  return x + y
end

# @param [Fixnum] x
# @ensures $result == x * x * x
def cube(x)
  return x * x
end
```

operators.rb

```
# @local [Fixnum] x
def main
  x = 0
  x += 5
  x -= 1
  x *= 2
  x /= 2
  x %= 2
  assert x == 0
end
```

stack.rb

```
# @ivar [Fixnum] size
# @ivar [Array<Fixnum>] elements
class Stack
  # @ensures @size == 0
  def initialize
    @size = 0
    @elements = []
```

```
  end

  # @ensures @size == old(@size) + 1
  def push(element)
    @size = @size + 1
    @elements.push element
  end

  # @requires @size > 0
  # @ensures @size == old(@size) - 1
  def pop
    @size = @size - 1
    @elements.pop
  end
end

# @local [Stack] stack
def use
  stack = Stack.new
  stack.pop
end
```