Transforming Source Code into Symbolic Transition Systems for Practical Model-Based Testing

Stan Janssen
stanjanssen@outlook.com

August, 2017, 35 pages

UvA supervisor: Clemens Grelck c.grelck@uva.nl

Host organisation: Axini http://axini.com
Host supervisor: Ivo Wever wever@axini.com

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

Abstract

Testing is a costly part of software development. Current methods of automated testing often require a significant amount of maintenance of test scripts. Axini B.V.* believes that model-based testing (MBT) solved that problem and has developed a tool, *TestManager*, to make MBT available to the industry. To do this, they offer the Axini Modelling Language (AML), a domain specific language developed in Ruby. A model written in AML can be compiled into a symbolic transition system (STS) representation of the system. This underlying formalism allows the model to specify both state and data based properties of systems. A finite domain solver is used to generate values for data-based properties.

Current research does not describe how to define properties on data in STS's beyond first order logic. This means it is difficult to express some properties in practice. One way to make it easier is to use Ruby functions instead. Unfortunately, the finite domain solver cannot solve for properties defined by Ruby functions. So no targeted test cases can be generated for those properties.

By transforming the Ruby function into AML, we can extend the STS by inlining the transformed function in place of the function call. This will allow the constraint solver to use the function for generating suitable test values. Another advantage of transforming a Ruby function into an AML model, is that the coverage tools used during simulation can keep track of what traces are covered inside the function.

*http://axini.com

Contents

Chapter 1

Introduction

Software testing is important to develop and maintain reliable software [2, 24]. If software is not adequately tested, the development and operating costs rise because of increased failures and increased rework. Common practices in software testing are manual and automated testing. The main drawback of manual testing is that it requires a significant effort to execute the same tests multiple times and is therefore expensive. Automated testing solves this problem by automatically executing test scripts, but brings additional costs upfront due to test script development and maintenance. [5, 23] Prior research states that testing can take up to 80% of the development budget [2].

Model-based testing (MBT) is a technique intended to reduce the costs of testing even further. MBT is a form of automated, specification based, black box, functionality testing. This means that the system under test (SUT) can only be observed and controlled via its external interfaces [14]. Using a model of the SUT's behaviour, we can generate, execute and evaluate test cases automatically. Testers only need to create and maintain the models. While MBT has a potentially higher cost upfront, the goal is to make it cheaper to maintain. Also, the MBT approach can be used to formalise a specification early in the development process, finding specification errors as early as possible.

Most MBT tools use state oriented formalisms that suffer from state space explosion, because of their explicit representation for the states of the specification [4]. Data oriented tools are specialised in data generation but lack the often required concept of state. Frantzen et al. proposed the symbolic transition system (STS) as a way to solve these problems by combining state and data oriented formalisms [14]. However, the language used to model properties of data in this formalism uses only a limited set of operations and types. Researchers have mostly focused on the modelling of observable systems, but when modelling more complex computations models often become difficult to comprehend.

1.1 Motivation

To bring MBT to the industry, Axini developed *TestManager*. TestManager is a tool which, among other functionality, generates test cases from a model written in the Axini Modelling Language (AML) and executes them on the SUT. AML is a domain specific language developed in Ruby by Axini. Models written in AML are compiled into STS representations of the system.

Modelling more complex computations in AML is done using *AML functions*. The bodies of these functions are written in Ruby and can, therefore, express computations more concisely. A Ruby interpreter executes the AML functions. The drawback is that constraints that use AML functions cannot be solved. Constraint solving is necessary to generate appropriate input for test cases.

Another drawback is that coverage tools cannot inspect which paths are taken within the functions. Coverage is used to evaluate the result of tests by measuring which parts of the specification have been validated by performed tests. If all tests pass, it is important to know whether the same test was run a thousand times or a thousand tests covered every possible path specified by the specification.

Our research aims to extend the STS formalism to improve the modelling of more complex computations by expanding the language used to specify properties of the modelled data. Important is that we can still use solvers to generate input values that satisfy these properties for targeted test cases.

1.2 Research questions

We use the following question as a guide to achieve our research goal:

> Is it possible to translate a Ruby function into an STS representation that effectively implements the same function, such that we can then use constraint solving to generate inputs for the function that will cover all traces through the function?

By "effectively implements the same function", we mean that for every input we provide the Ruby function and the STS, they both return the same output. To answer this question, we will answer the following subquestions:

1. What relations between Ruby and STS's can we identify that help us in making such translations?
2. For what subset of Ruby can we perform such translations?
3. Can the finite domain solver solve the constraints in the resulting STS's such that we can indeed generate inputs that will cover all possible traces through the function?

1.3 Contributions

In this thesis, we investigate which concepts of the Ruby programming language can be transformed to (combinations of) STS concepts. The result is a proof of concept compiler which decreases the complexity of modelling in AML and increases the usability of the models pertaining to complex computational behaviour. The compiler compiles a Ruby function definition into a valid AML model. The design of the compiler is described using compilation schemes. Additionally, we discuss an empirical method to show the validity of the compilation schemes. Our work could also be used as a basis for the model checking of Ruby source code.

1.4 Overview

This document consists of five chapters. We first present the foundations of our work in Chapter 2. The implementation of our compiler is discussed in Chapter 3. Chapter 4 contains information on the process of validating the models generated by our compiler. In Chapter 5 we discuss related work and finally, in Chapter 6, we summarise our work and discuss our findings and future work.

Chapter 2

Foundations

2.1 Model-based testing

Software testing consists of four steps: (1) specification interpretation, (2) test creation, (3) test execution and (4) test result evaluation. In manual testing, all these steps are done by hand. In automated testing, the test execution and result evaluation are done automatically with the use of test scripts. The test scripts are created by a tester who interprets the specification, talks to designers of the system and interviews the end users. Therefore the quality of testing depends on the skill of the tester as well as the quality of specifications and discussions.

MBT (model-based testing) takes the automation one step further by automating test case creation as well. Instead of creating many test scripts, the tester creates formal models of the behaviour of the system under test (SUT). Practically speaking, a model is sufficiently formal if it can be manipulated automatically for test case generation. [6]

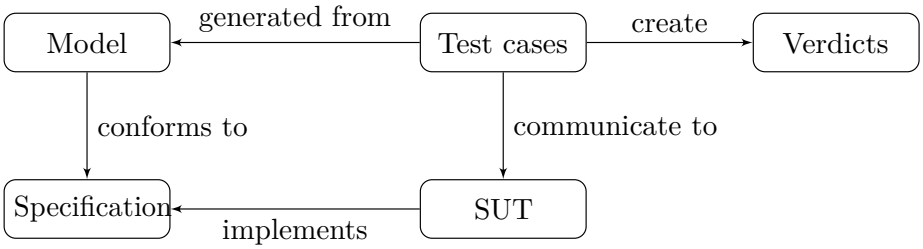

Figure 2.1: The MBT pipeline.

The process of MBT begins with the creation of a model conform the specification (see Figure 2.1). The tester develops a system-readable version of the intended behaviour of the system. To ensure the same test results given a model, SUT and environment, the model should be unambiguous. To achieve this, one of two formalisms is commonly used: finite state machines [3, 10, 20] or labelled transition systems (LTS's) [11, 26]. Because LTS theory is the basis for symbolic transition systems (STS's) [14, 15], which our work builds upon, we will only introduce the latter formalism.

Once the model is defined, we can algorithmically generate test cases using information from the model and an implementation relation. An implementation relation defines the formal conditions under which the implementation conforms to the specification [25]. Because of the mathematical nature of the model and the implementation relation, it can be proven that the test cases are sound [26].

Using the generated test cases, we can then test the system. Usually, an adapter is used to decouple knowledge of the SUT's external interfaces from the test generation tool. The test cases provide the inputs and expected outputs to the adapter. The adapter forwards the inputs to the SUT. When the SUT sends a response, the adapter observes it and evaluates it using the expected outputs.

For the evaluation of the test results, coverage information is tracked through the transition system. The coverage information is then used to evaluate which parts of the SUT under which conditions fail.

2.2 Labelled transition systems

A commonly used formalism in MBT is the LTS, more specifically the input-output LTS. An LTS is a collection of states with transitions between them. The transitions specify either input or output behaviour. We refer to the paper by Tretmans for a formal description of LTS's [26].

When a system uses inputs and outputs to communicate with its environment, an input/output LTS can be used to model its behaviour. That is, when the LTS is in the source state of an input transition, the SUT expects an input. Similarly, when the LTS is in the source state of an output transition, an output can be observed from the SUT. Upon sending an input or observing an output, the LTS advances to the next state. [9] For test case generation with input/output LTS's, the implementation relation *ioco* (as defined in Definition 1) can be used.

Definition 1 (Input/Output Conformance (*ioco*))**.** An implementation is *ioco* to its specification when, for all traces, i.e. sequences of observable actions [6], the implementation output is a subset of the specification output. In other words, the output of the developed system should be a subset of the expected output. This relation is formally described in Equation 2.2.

$$i \ ioco \ s =_{def} \forall \sigma \in Straces(s) : out \ (i \ after \ \sigma) \subseteq out \ (s \ after \ \sigma) \tag{2.2}$$

Where

- i is the implementation;
- s is the specification;
- σ is a trace in the transition system;
- $Straces(s)$is the set of traces in the specification;
- $out(n \ after \ \sigma)$ is the output as observed from a model n after following trace σ.

2.3 Symbolic transition systems

A modern formalism that can be used to model the SUT in MBT is the STS. This formalism was the result of research into overcoming state space explosion and information loss problems that you run into with LTS's when modelling data [14]. The implementation relation for STS's is derived from input/output conformance (*ioco*), an implementation relation used for LTS's. A formal definition of STS's is shown in Definition 2.

Definition 2. (Symbolic Transition System [15])

An STS is a tuple $\langle \mathcal{S}, \sigma_0, \mathcal{V}, \mathcal{P}, \mathcal{L}, \rightarrow \rangle$ where

- $\mathcal{S}$ is a set of states and σ_0 is the initial state;
- $\mathcal{V}$ is a set of state variables. These variables are global to the STS and used as data storage;
- $\mathcal{P}$ is a set of label parameters ($\mathcal{V} \cap \mathcal{P} = \emptyset$), used only on a single transition;
- $\mathcal{L}$ is a set of observable transition labels. A label is observable when it represents input or output behaviour of the SUT;
- $\rightarrow$ is the transition relation. Each transition is labelled with either a label from $\mathcal{L}$ or τ in case the transition is unobservable. We refer to these unobservable transitions as τ-transitions. Transitions can have a constraint and an update mapping. Constraints are defined using variables from $\mathcal{V}$ and $\mathcal{P}$. Update mappings are assignments to variables in $\mathcal{V}$. A transition can only be followed if the constraint is satisfied. When a transition is followed, the update mapping is applied. We write $\sigma \xrightarrow{\lambda,\varphi,\rho} \sigma'$ to describe the transition from state σ to σ' with label λ, constraint φ and update mapping ρ.

The visualisation of a train ticket selling machine in Figure 2.3 shows six states and seven transitions. Input and output labels are denoted by a question mark ('?') and exclamation mark ('!'), respectively. Variables are printed in italics (e.g. *total*), label parameters start with an underscore (e.g. *_coin*).

A trace could start with selecting "single" on the machine ($s_0 \xrightarrow{?select\_single,\top,total=4} s_1$). This sends the stimulus '`select_single`' to the machine and sets the state variable '`total`' to 4. The τ-transition from state 2 to state 3 is automatically followed. We then insert a 2 euro coin into the machine. The

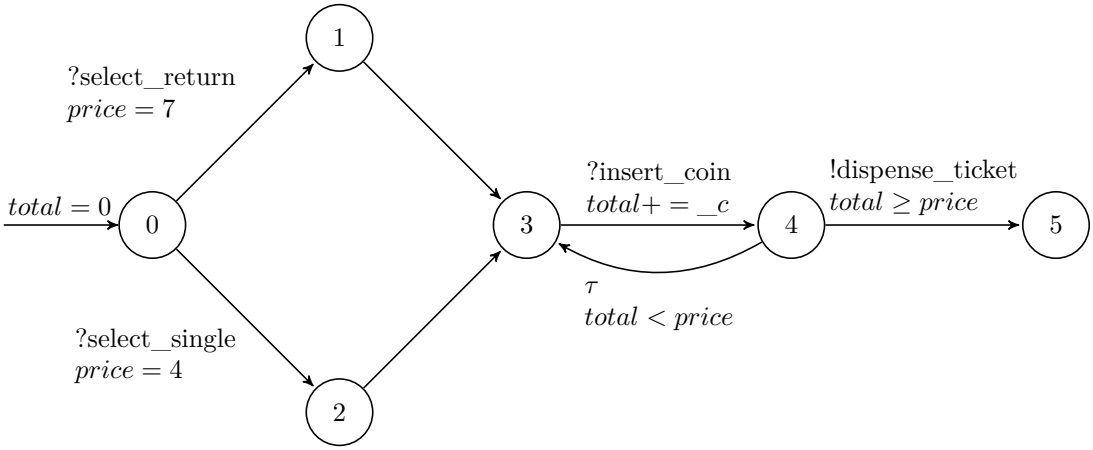

Figure 2.3: STS of a ticket selling machine.

transition from 3 to 4 updates our '`total`' from 0 to 2. Since our '`total`' (2) is not greater or equal to the price of the ticket (4), the constraints force us to follow the τ-transition from 4 back to 3. After inserting another 2 euro coin, the '`total`' is updated from 2 to 4 and the machine sends a response (dispensing a ticket) in the transition from 4 to 5.

2.4 Axini Modelling Language

To use the STS theory discussed in Section 2.3, we need to be able to model the actual SUT on a computer and then interpret this model. This section will discuss how MBT is applied at Axini using their modelling language.

Systems are modelled using the Axini Modelling Language (AML). By interpreting a model, it can simulate traversal of the STS. By using the finite domain solver in GNU Prolog[1] to solve constraints on transitions, the software can generate test cases by selecting a trace and solving the constraints along it. Appendix B shows a complete reference of all the concepts in AML referenced in this thesis.

Train ticket selling machine example.

Figure 2.4 shows the AML representation of the STS in Figure 2.3. On line 1, we create an internal channel with the name *interaction*. An external channel is a communication channel between the SUT and its environment. The SUT does not know it is being operated by the test framework, so the channel enables the STS to behave like the environment.

Line 3 starts the description of an STS named '`tickets`'. The STS uses the '`interaction`' channel to receive three stimuli and send one response. The '`insert_coin`' stimulus declares one parameter named '`coin`' with type '`float`'.

On line 10 and 11 we declare state variables. The variable '`total`' is immediately initialised with a value of 0.0.

Line 13 creates multiple paths in the STS. The simulator can take one of the options, denoted by '`o`' on lines 14 and 17, randomly. After the '`choice`', the STS can only receive the '`insert_coin`' label. Upon receiving this label, it updates the '`total`' variable.

Finally, the STS repeats picking one of the two options declared on lines 24 and 28. The simulator can choose an option based on which constraint it can fulfill. In this case, while the '`total`' is smaller than the '`price`', it expects to receive the '`insert_coin`' label. Once the '`total`' is big enough, it chooses the other option and break out of the '`repeat`' construct.

[1] `http://gprolog.org/`

```
internal 'interaction'

process('tickets') {
  channel('interaction') {
    stimulus 'select_single'
    stimulus 'select_return'
    stimulus 'insert_coin', '_coin' => :float
    response 'dispense_ticket'
  }

  var 'price', :int
  var 'total', :float, 0.0

  choice {
    o {
      receive 'select_single', update: 'price = 4'
    }
    o {
      receive 'select_return', update: 'price = 7'
    }
  }

  receive 'insert_coin', update: 'total = total + _coin'
  repeat {
    o {
      constraint 'total < price'
      receive 'insert_coin', update: 'total = total + _coin'
    }
    o {
      constraint 'total >= price'
      send 'dispense_ticket'
      stop_repetition
    }
  }
}
```

Figure 2.4: AML representation of the ticket selling machine.

STS simulation. Simulation of the STS is done using a simulator similar to the STSimulator [13]. The simulator keeps track of the current state and the transitions that can be followed. The simulator can be used to verify whether a trace with specified label parameters can be executed. It can also be used to compute the values for label parameters in order to execute a trace. These values need to be chosen such that the transition constraints are satisfied.

When the simulator is used to test a system, it acts like the user of the system. This means that it observes the output of the system and generates the input given to the system. Simulation of the SUT is nondeterministic. When the model contains ambiguous states due to the lack of certain details, the simulator cannot know in which state it currently is in.

Uncertainty about the current state occurs when the simulation is in a source state of multiple transitions with the same label. The simulator advances on all possible transitions and keeps track of the states the simulation could currently be in. Only when the SUT sends a response that is only specified in one of the tracked states does the simulator know in which state the simulation exactly is.

For example, in Figure 2.5, if we provide 5 as input to the SUT then both constraints are satisfied. In this case, the simulator does not know in which state the SUT is. The simulation continues in state 1, 2 and 3. If the SUT responds with !*exit*, the simulator knows the SUT was actually in state 2. Similarly when the simulator observes the output !*one* or !*three* it knows the SUT was in state 1 or 3, respectively.

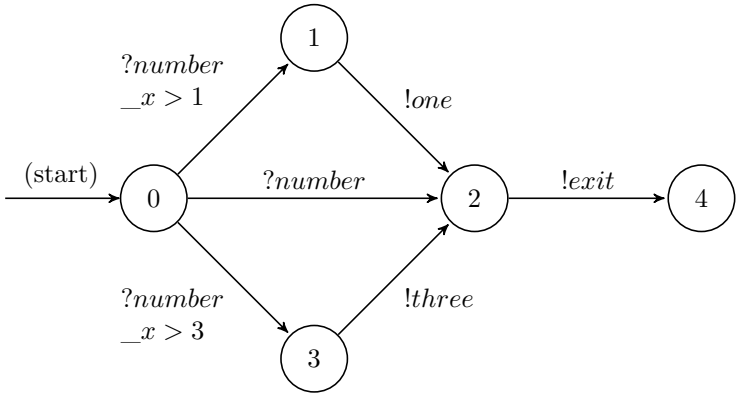

Figure 2.5: Example of nondeterministic behaviour in the simulation of STS's.

Constraint language. Constraints and update mappings on transitions are expressed using a set of operations and a set of types. The types are *string, integer, boolean, decimal, date* and a homogeneous list of these types (e.g. `[:string]` is a list of strings). It is also possible to create a combination of types by creating a *struct*. A struct is a collection of name-type pairs surrounded by braces, e.g. `{'name' => :string, 'age' => :integer}` . The set of operations is defined as follows:

- Arithmetic operators
  addition, subtraction, multiplication, division and *modulo*.
- Boolean operators
  and, or, not
- Relational operators
  equals, greater than, less than and *subset*.
- Miscellaneous operations
  assignment, function call, list/struct creation, list/struct indexing, length, max and *min*.

These operations are chosen by the developers of AML because they map nicely to Prolog queries. The mapping with Prolog is important as the finite domain solver in Prolog is used to generate input for targeted test cases.

However, the use of this limited set of types and operations makes it difficult to model complex systems. Complex operations require a large sequence of the supported operations. An analogy for this problem is to write a Java program in Assembler. Programs that use classes can be written in Assembler but require a significantly greater amount of code.

To prevent models from becoming verbose, AML supports the definition of functions where Ruby is used. Ruby has a larger set of types and operations and can express complex operations more concisely. A drawback of these functions is that the simulator cannot use them to compute values to satisfy constraints as it is not written in the constraint language.

To use AML functions, a type signature needs to be defined. This signature indicates what the types of the input arguments and return value are. Once defined, the function can be used with label parameters and state variables in updates on input/output transitions and constraints on output transitions. More information on AML functions is given in Appendix B.

Chapter 3

Ruby-AML Compiler

In this chapter we explain how to implement the Ruby to AML (the Axini Modelling Language) compiler. To do this, we first need to understand what the input to our compiler is and what the properties of our output are. Following this is the discussion of the different parts of the compiler and the relation between different Ruby and AML concepts.

3.1 Input/output specification

The input to our compiler is the source code of a valid Ruby function definition and the AML types of the function parameters. Function definitions start with the '`def`' keyword, followed by a function name and optionally any parameters. The definition ends with the '`end`' keyword. In between the '`def`' and '`end`' keywords are Ruby expressions. The set of supported expressions is limited to the subset as defined by the grammar in Figure 3.2. The output is an AML model that outputs the same value as the Ruby function would return.

The compiler is a pipeline with five components (see Figure 3.1). First the *parser* parses the Ruby source code and outputs an abstract syntax tree (AST). Secondly, using the AST, the *checker* verifies that the input only contains Ruby from the defined subset. Thirdly, the *rewriter* desugars the input, converting complex Ruby concepts into simpler ones. Fourthly, the type *propagator* derives the types of all local Ruby variables from the input signatures. Finally, the *code generator* transforms the rewritten source code into AML. In this section, we discuss the functionality of each of these components.

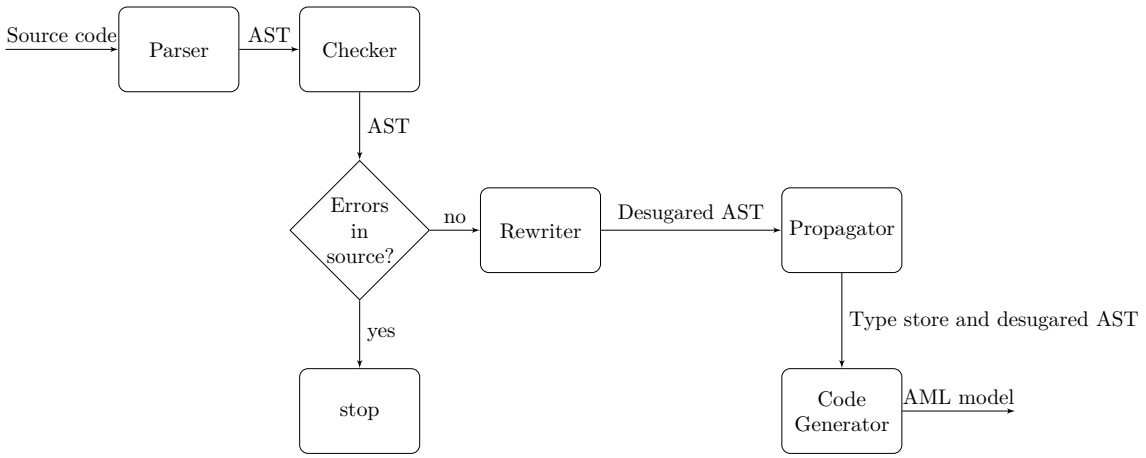

Figure 3.1: Pipeline displaying the different components of the compiler.

```
Value        :  integer | decimal | string | boolean | date | nil | variable

Expr         :  Expr binop Expr
             |  unop Expr
             |  ( Expr )
             |  Expr [ Expr ]
             |  [ [Value [, Value ]* ] ]
             |  { [Expr ⇒ Expr [, Expr ⇒ Expr ]* ] }
             |  MethodCall | FunctionCall | Value

FunctionCall :  functionName ( [Expr [, Expr ]* ] )

MethodCall   :  Expr . FunctionCall

ControlFlow  :  if Expr then [Stmt ]+ [ elsif Expr then [Stmt ]+ ]* [ else [Stmt ]+ ] end
             |  loop do [Stmt ]* end
             |  while Expr do [Stmt ]* end
             |  for variable in Expr do [Stmt ]* end

Stmt         :  variable = Expr
             |  FunctionCall | MethodCall
             |  ControlFlow
             |  return Expr

FunctionDef  :  def functionName [ ( [ variable [, variable ]* ] ) ] [Stmt ]* end
```

Figure 3.2: BNF grammar for the Ruby subset as supported by the code generator. Symbols in bold are literal text, Words in italics are non-terminals and regular words are terminals (e.g. 'binop' and 'unop' are placeholders for binary and unary operators, respectively). The individual *Value*s are recognised by the lexer.

3.2 Parser

To compile Ruby, we use the AST of the function definitions as the input to our compiler. The AST contains different types of nodes for different syntactic elements. Figure 3.3 shows an example for the `if/else` statement. In this figure, `:lvar` denotes a reference to a local variable and `:begin` is a parent node for a sequence of statements.

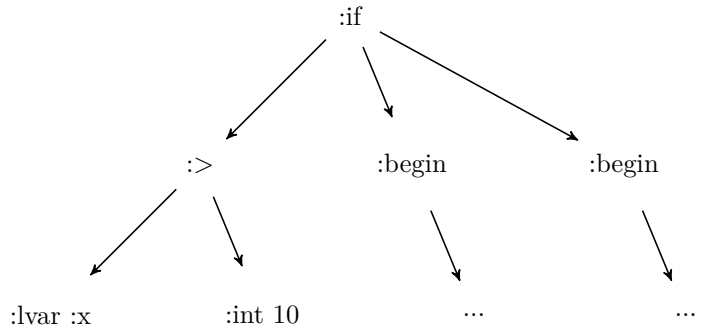

Figure 3.3: Simplified Ruby AST of an `if/else` statement.

To obtain the AST from Ruby source code we use Peter Zotov's *parser*[1] library. Using this parser, we retrieve the AST from Figure 3.3 in the format shown in Figure 3.4. Each node is in the format of `s(type, child_0, child_1, ...)` .

```
s(:if,
  s(:send, s(:lvar, :x), :>, s(:int, 10)), # condition
  s(:begin, ...),                          # statements if condition is met
  s(:begin, ...)                           # statements if condition is not met
)
```

Figure 3.4: Ruby AST as output by the *parser* library.

Then, using Zotov's *AST*[2] library, we can process this tree node by node. An AST processor has a method for every type of node with the name '`on_<type>`'. Figure 3.5 shows an example of how we process an `if` node using the '`on_if`' method.

```
1  def on_if(node)
2    condition, statements, elsestatements = *node
3    choice = Choice.new
4
5    condition_string = process(condition)
6
7    if_option = create_option(condition_string, statements)
8    else_option = create_option("!(#{condition_string})", elsestatements)
9    choice.add_options(if_option, else_option)
10   choice
11 end
```

Figure 3.5: Method to process an `if` node.

In this example, we first retrieve the three children of the node on line 2. We then process the condition node to get it as a string on line 5. On lines 7 and 8, we create two options for the AML choice. The simulator only picks the first option, if the `if` condition is satisfied. For the `else` option, we need to invert the constraint; otherwise, the simulator also picks this option even when the `if` condition is satisfied. Finally, we add the options to the `choice` object and return it.

3.3 Checker

Before applying any transformation, the checker ensures the input contains only concepts included in the supported subset of Ruby. Appendix A describes the different types of errors returned by the checker. As the AST contains source location information in every node, we can include this information in the errors too. This location information is useful for debugging purposes, as it allows the developer to see which expression on which line causes the error.

Unsupported expressions. The checker is a type of AST processor. For every node which has a '`on_<type>`' method defined it traverses the children. When it encounters a node for which the processor has defined no method, it adds an error.

Conditionally supported expressions. Our compiler does not support some expressions inside all other expressions where Ruby allows them to occur. For example, in Ruby, it is possible to define another function inside a function definition. Our compiler supports function definitions, but only at the root level of the AST.

To keep track of which expression we are in, we keep a stack of expression types. When we enter a new expression, we add the type to the top of the stack. When the checker has completed checking the expression, we pop it from the stack. Checking if we are inside a certain expression is then done by looking at every item on the stack to see if it matches.

[1] https://github.com/whitequark/parser
[2] https://github.com/whitequark/ast

Example. An example of input that will yield an error is shown in Figure 3.6. The error returned is as follows:

```
ERROR   (2:6)   Unsupported recursion. Only tail recursion is supported.
    Source: 'foo(x + 1)'
```

The message consists of two lines. The first line contains the line and column number where the error was triggered (‘2:6’) and the error type. The faulty input itself is shown on the next line.

```
1 def foo(x)
2   x = foo(x + 1) if x < 10
3   x = x * x
4   return x
5 end
```

Figure 3.6: On this input, the checker will return ‘Unsupported recursion. Only tail recursion is supported.’.

3.4 Rewriter

Certain Ruby concepts can be *desugared*, i.e. rewritten as basic concepts. Before the compiler compiles the input to AML, the rewriter rewrites any concept that can be desugared into our base subset (see Figure 3.2).

Concepts that are desugared are: operator assignment, simultaneous assignment, ‘unless’, ‘until’, guard clauses, ternary operators and ‘each’. As desugaring is a thoroughly researched area [8, 17, 21], we do not discuss the inner workings of the rewriter here.

3.5 Propagator

Because AML requires variables to be declared with a type, we need to derive the types of all local Ruby variables. In this section, we discuss how types in the input source are inferred using induction on the subset grammar (Figure 3.2). The induction systems use the following notation:

- the proposition below the line (conclusion) is true given that all the propositions above the line (premises) hold;
- $x : xs$ is a list with x as its first element and xs as its the non-empty tail.
- Γ is a type store, such that $\Gamma(x) \rightarrow \begin{cases} type & \text{if } \langle x, type \rangle \in \Gamma. \\ \bot & \text{otherwise} \end{cases}$

The propagator is initialised with the AST of the input source and a type store containing key-value pairs for the types of the input arguments and optionally the type of the return value. The type signatures of functions and methods used in the source code are extracted from the source code using the grammar listed in Figure 3.7 and stored in the type store. It then traverses the AST to infer the types of the variables. Because Ruby requires a variable to be initialised before it is used, the type store always contains its type when the propagator encounters it in an expression.

The parser provides the right types for any of literal *Value*s. So whenever the propagator encounters a value, it implicitly knows the type. The type inference for expressions (*Expr*) is shown in Figure 3.9-3.13.

Type : **:integer** | **:decimal** | **:string** | **:boolean** | **:date** | **:void** | **:symbol**
| [*Type*]
| **{** *Type* ⇒ *Type* **}**
| **{** identifier ⇒ *Type* ⌈, identifier ⇒ *Type* ⌋* **}**

Signature : # functionName :: [⌈*Type*, ⌈, *Type* ⌋* ⌋] ⇒ *Type*

Figure 3.7: BNF grammar for type signatures.

Operators	Left-hand side	Right-hand side	Output
+	date	integer	date
	integer	integer	integer
	integer | decimal	decimal	decimal
	decimal	integer	decimal
	string	string	string
-	integer	integer	integer
	decimal	decimal	decimal
*, /, %	integer	integer	integer
	integer | decimal	decimal	decimal
	decimal	integer	decimal
&&, ||	boolean	boolean	boolean
! (not)		boolean	boolean
= (assign)	any	left-hand side	left-hand side
	any	void	left-hand side
	decimal	integer	decimal
$=, \neq$	any	left-hand side	boolean
	any	void	boolean
	void	any	boolean
	integer	decimal	boolean
	decimal	integer	boolean
$>, \geq, <, \leq$	date	date	boolean
	integer	integer	boolean
	integer | decimal	decimal	boolean
	decimal	integer	boolean
set equal, $\in$	[any]	left-hand side	boolean

Figure 3.8: Type signatures of operators in AML. *left-hand side* inside a cell denotes that the the type in this cell has to be equal to the left-hand side during runtime.

Binary operators have a left-hand side and right-hand side expression as arguments. **Unary operators** only contain a single expression as an argument. Figure 3.8 shows what the resulting type of an expression containing a binary and unary operator is given the types of the arguments for the operator. Figure 3.9 shows how the propagator uses this table.
An expression in **parentheses** has the same type as the expression without parentheses (Figure 3.10).

Lists in AML must be homogeneous, i.e. contain elements with identical types. The propagator infers the type if the type of the first element is the same as the type of the tail of the list. A singleton list is by definition homogeneous, and its type is thus the type of the only element. The type of an empty list is *void*, because there are no elements to infer the type from. Void is a special type which is considered a valid substitute for the other types when it comes to return values. The type inference for these three types of lists is shown in Figure 3.11.

$$\frac{\langle Expr_1, \Gamma\rangle \to t \qquad \langle Expr_2, \Gamma\rangle \to u \qquad \langle \oplus, t, u\rangle \in Table}{\langle Expr_1 \oplus Expr_2, \Gamma\rangle \to Table[\![\oplus, t, u]\!]}$$

$$\frac{\langle Expr, \Gamma\rangle \to t \qquad \oplus \in Table.Operators}{\langle \oplus Expr, \Gamma\rangle \to Table[\![\oplus, \emptyset, t]\!]}$$

Figure 3.9: Type inference for binary and unary operators. *Table* refers to Figure 3.8 and $Table[\![op, t_1, t_2]\!]$ is a look-up for the output type of operator *op* with argument types t_1 and t_2.

$$\frac{\langle Expr, \Gamma\rangle \to t}{\langle \text{ (} Expr \text{) }, \Gamma\rangle \to t}$$

Figure 3.10: Type inference for the expressions enclosed in parentheses

1. $$\frac{}{\langle \text{ [ ] }, \Gamma\rangle \to void}$$

2. $$\frac{\langle Expr, \Gamma\rangle \to t}{\langle \text{ [} Expr \text{] }, \Gamma\rangle \to [t]}$$

3. $$\frac{\langle Expr, \Gamma\rangle \to t \qquad \langle Rest, \Gamma\rangle \to [t]}{\langle Expr : Rest, \Gamma\rangle \to [t]}$$

Figure 3.11: Type inference on lists. 1. Empty list, 2. Singleton list, 3. Multi-element list.

The collection of key-value pairs can either be a type-type mapping (e.g. integers to strings) or key-value mapping (i.e. strings to any type). Figure 3.12 shows how the propagator handles these two types. **(1)** The type of a single key-value mapping is the types of its key and value. **(2)** If all the pairs have a key of type t and a value of type u, then the collection is a type-type mapping with type $\{t \Rightarrow u\}$. **(3)** If all the keys are strings, but the values are of different types then the collection is a key-value mapping (type $\{string \Rightarrow any\}$). **(4)** AML expects a more specific type, namely a collection of the actual key names and the type of their values: $\{key_0 \Rightarrow t_0, key_1 \Rightarrow t_1 ...\}$. To do this, a second iteration over the expression is needed where the type is updated from $\{string \Rightarrow any\}$ to the more specific $\{key \Rightarrow type ...\}$ type. In the formalisation, the second iteration is denoted as $\langle ... \rangle'$.

1. $$\frac{\langle E_1, \Gamma\rangle \to t \qquad \langle E_2, \Gamma\rangle \to u}{\langle \text{ \{ } E_1 \Rightarrow E_2 \text{ \} }, \Gamma\rangle \to \{t \Rightarrow u\}}$$

2. $$\frac{\langle E_1, \Gamma\rangle \to t \qquad \langle E_2, \Gamma\rangle \to u \qquad \langle Rest, \Gamma\rangle \to \{t \Rightarrow u\}}{\langle \text{ \{ } E_1 \Rightarrow E_2 : Rest \text{ \} }, \Gamma\rangle \to \{t \Rightarrow u\}}$$

3. $$\frac{\langle \text{ \{ } E_1 \Rightarrow E_2 \text{ \} }, \Gamma\rangle \to \{string \Rightarrow t\} \qquad \langle Rest, \Gamma\rangle \to \{string \Rightarrow u\} \qquad t \neq u}{\langle \text{ \{ } E_1 \Rightarrow E_2 : Rest \text{ \} }, \Gamma\rangle \to \{string \Rightarrow any\}}$$

4. $$\frac{\langle \text{ \{ } E_1 \Rightarrow E_2 : Rest \text{ \} }, \Gamma\rangle \to \{string \Rightarrow any\} \qquad \langle \text{ \{ } E_1 \Rightarrow E_2 \text{ \} }, \Gamma\rangle' \to \{k \Rightarrow t\}}{\langle \text{ \{ } E_1 \Rightarrow E_2 : Rest \text{ \} }, \Gamma\rangle' \to \{k \Rightarrow t\} \cup \langle \text{ \{ } Rest \text{ \} }, \Gamma\rangle'}$$

Figure 3.12: Type inference on key-value pairs. *Expr* is abbreviated to *E*

The types of **function and method calls** are defined by their return types. Because the type signatures in the source code provide the return types, the propagator can use its type store to return the appropriate type (Figure 3.13).

$$\frac{}{\langle Call, \Gamma \rangle \rightarrow \Gamma(Call)}$$

Figure 3.13: Type inference for function and method calls.

3.6 Code generator

In this section we discuss the concepts that our code generator supports using compilation schemes.

To compile from Ruby to AML we need to know the types of variables provided by the propagator. During compilation we need to keep track of which variables have already been declared, because AML does not allow variables to be declared more than once.

Unary and binary expressions in AML as listed in Figure 3.8 are identical in Ruby. Therefore the compilation only recurses on the subexpression(s). Literals and variables in Ruby are also identical in AML and are therefore left as they are (Figure 3.14). Figure 3.15 shows the compilation scheme for unary and binary expressions.

$$\begin{array}{lcl} \mathcal{C}[\![literal]\!] & \Rightarrow & literal \\ \mathcal{C}[\![variable]\!] & \Rightarrow & variable \end{array}$$

Figure 3.14: Compilation schemes for literals and variables.

$$\begin{array}{lcl} \mathcal{C}[\![Expr_1 \oplus Expr_2]\!] & \Rightarrow & \mathcal{C}[\![Expr_1]\!] \ \oplus \mathcal{C}[\![Expr_2]\!] \\ \mathcal{C}[\![\oplus Expr_1]\!] & \Rightarrow & \oplus \ \mathcal{C}[\![Expr_1]\!] \end{array}$$

Figure 3.15: Compilation schemes for unary and binary expressions where $\oplus \in Table.Operators$ (Figure 3.8).

The methods '`length`' and '`include?`' in Ruby are also supported in AML by using respectively '`length`' and '`in`'. In Ruby '`count`' and '`size`' are aliases for '`length`' and can be compiled identically. The compilation scheme of these particular methods is shown in Figure 3.16.

$$\begin{array}{lcl} \mathcal{C}[\![Expr.\text{length}]\!] & \Rightarrow & \textbf{length(}\ \mathcal{C}[\![Expr]\!]\ \textbf{)} \\ \mathcal{C}[\![Expr_1.\text{include? } Expr_2]\!] & \Rightarrow & \mathcal{C}[\![Expr_2]\!]\ \textbf{in}\ \mathcal{C}[\![Expr_1]\!] \\ \mathcal{C}[\![Expr_1\ [\ Expr_2\ ]]\!] & \Rightarrow & \mathcal{C}[\![Expr_1]\!]\ [\ \mathcal{C}[\![Expr_2]\!]\ ] \end{array}$$

Figure 3.16: Compilation schemes for the '`length`' and '`include?`' methods and array/struct indexing.

Local variable assignment in Ruby is used to associate a value with a name in the current scope. There are three scopes in Ruby: global, instance and local. A scope is used to access variables at different levels in a program. Global scope can be accessed anywhere. Instance scope can only be accessed anywhere within its namespace. Local scope can only be accessed in the current function definition. Because our compiler receives a function definition as input, we only need to consider the local scope. Global scope could be accessed during runtime, but since we compile every function definition individually variables in the global scope are not accessible during compilation.

The compilation scheme for variable assignment in Ruby is, shown in Figure 3.17, shows two parts. The first part ensures the variable is declared in the top section of the model. In case the variable is already declared, we do not declare it again. The second part assigns a value to the variable by compiling the expression behind the assignment operator.

$$\mathcal{C}[\![Rest_1 \ ; \ variable = Expr \ ; \ Rest_2]\!] \quad \Rightarrow \quad \mathcal{D}[\![variable = Expr]\!] \ \mathcal{C}[\![Rest_1]\!] \ \textbf{update} \ " \ variable = \mathcal{C}[\![Expr]\!] \ " \ \mathcal{C}[\![Rest_2]\!]$$

$$\mathcal{D}[\![variable = Expr]\!] \quad \Rightarrow \quad \textbf{var} \ \ variable \ \ , \ \Gamma(variable) \ \ \|\textit{if variable not declared otherwise raise error}$$

Figure 3.17: Compilation schemes for local variable assignment.

Function and method calls are compiled to separate AML functions to perform just that call. This means that constraints using the Ruby calls can no longer be solved by the finite domain solver. However, by separating these calls, we can still solve constraints on other parts of the trace.

To define an AML function, a type signature is required. Type signatures should be added in the source code following the grammar in Figure 3.7.

Figure 3.18 shows the compilation schemes for function calls. First the AML function is created, then the function call is put into the right place with compiled arguments. Turning a function call into an AML function creates an AML function definition near the top of the model with the same name as the original function. The arguments are renamed to _0, _1 etc. The body of the AML function is a call to the original function with the renamed arguments.

$$\mathcal{C}[\![Rest_1 \ foo \ (\ Exprs \ ) \ \ Rest_3]\!] \quad \Rightarrow \quad \mathcal{F}[\![foo \ (\ Exprs \ )]\!] \ \mathcal{C}[\![Rest_1]\!] \ \ foo \ (\ \mathcal{C}[\![Exprs]\!] \ ) \ \ \mathcal{C}[\![Rest_3]\!]$$

$$\mathcal{F}[\![foo \ (\ Exprs \ )]\!] \quad \Rightarrow \quad \textbf{function(} \ " \ \ foo \ \ " \ , \Gamma(foo) \ ) \ \{ \ | \ \ \mathcal{G}[\![Exprs]\!] \ [\![0]\!] \ \ | \ \ foo \ (\ \mathcal{G}[\![Exprs]\!] \ [\![0]\!] \ ) \ \} \ \ \|\textit{if function not defined}$$

$$\mathcal{C}[\![Expr \ : \ Rest]\!] \quad \Rightarrow \quad \mathcal{C}[\![Expr]\!] \ , \mathcal{C}[\![Rest]\!]$$

$$\mathcal{G}[\![Expr \ : \ Rest]\!] \ [\![count]\!] \quad \Rightarrow \quad \_count, \mathcal{G}[\![Rest]\!] \ \ [\![count + 1]\!]$$

$$\mathcal{G}[\![[Expr]]\!] \ [\![count]\!] \quad \Rightarrow \quad \_count$$

$$\mathcal{G}[\![[\ ]]\!] \ [\![count]\!] \quad \Rightarrow$$

Figure 3.18: Compilation schemes for function calls.

Method calls are similar to function calls. The two differences are that **(1)** the name of the AML function is prefixed with the type of the receiver expression (i.e. the expression whose method is called) to support method overloading and **(2)** that the receiver is passed as the first argument. This is shown in Figure 3.19.

$$\mathcal{C}[\![Rest_1 \ Expr \ . \ foo \ (\ Exprs \ ) \ \ Rest_3]\!] \quad \Rightarrow \quad \mathcal{F}[\![Expr \ . \ foo \ (\ \ Exprs \ )]\!] \ \mathcal{C}[\![Rest_1]\!] \ \ foo \ (\ \mathcal{C}[\![Expr \ : \ Exprs]\!] \ ) \ \ \mathcal{C}[\![Rest_3]\!]$$

$$\mathcal{F}[\![Expr \ . \ foo \ (\ Exprs \ )]\!] \quad \Rightarrow \quad \textbf{function(} \ " \ \ \langle Expr, \Gamma \rangle \ \ \_\_ \ \ foo \ \ " \ , \Gamma(\_foo) \ ) \ \{ \ | \ \ \mathcal{G}[\![Expr \ : \ Exprs]\!] \ [\![0]\!] \ \ | \ \ \textbf{\_\_0.} \ foo \ (\ \mathcal{G}[\![Exprs]\!] \ [\![1]\!] \ ) \ \} \ \ \|\textit{if method not defined}$$

Figure 3.19: Compilation schemes for method calls.

Using the propagator to derive the type of the expression, an AML function is created with the name '`<type>_foo`' and the type signature for the function is looked up in the type store. In the body

we use the first argument to refer to the original expression whose method was called and call the method using the renamed arguments. In place of the original method call, the model calls the newly defined AML function prefixing the expression to the list of arguments. Figure 3.20 gives a concrete example.

```
1  a = 6
2  b = 9
3  # :to_s :: [:integer] => :string
4  x = a.to_s
5  # :gcd :: [:integer, :integer] => :integer
6  y = b.gcd(a).to_s
7  # :to_s :: [:float] => :string
8  z = 3.4.to_s
```

Figure 3.20: Example of calling methods on primitives.

In this example, we specify that '`a.to_s`' takes an integer and returns a string (line 3). Line 6 shows that it is possible to chain methods. Because, the signature of '`Integer#gcd`' is not defined yet, we need to define it first on line 3. In addition to the receiver ('`b`'), it takes another integer as an argument and then returns an integer. Because we defined '`to_s`' for integers on line 3, we do not need to define it again. If the source code includes an overloaded method, we need to specify the output type (lines 7, 8) again.

Control flow allows the program to take a different path or repeat a path through the program depending on the result of an expression. In our subset of Ruby, control flow is done with '`if/elsif/else`' expressions, guard clauses and ternary operators (see Figure 3.21). The rewriter has taken care of rewriting the guard clauses and ternary operators as '`if/else`' expressions, so our code generator only need to compile '`if/else`' and '`if/elsif/else`' expressions. Figure 3.22 shows the compilation schemes for these expressions.

```
1  # guard clause
2  x = 1 if y > 3
3
4  # ternary operator
5  x = y > 3 ? 1 : 0
6
7  # if/elsif/else
8  if y > 3
9    x = 1
10 elsif y < 0
11   x = 3
12 else
13   x = 0
14 end
```

Figure 3.21: Three different types of control flow in Ruby.

As the schemes show, the code generator handles '`elsif`' expressions as nested '`if`' expressions. This is not strictly necessary from a transition system perspective as multiple constraints can be joined conjunctively on a transition. However, it makes the implementation of the compiler more uniform as there is one variant less to consider. Figure 3.23 shows an alternative compilation scheme where only a single AML '`choice`' statement is used with an option for every condition and the else.

As the *else clause* is empty in the rewritten *guard expression*, the code generator has one option with the originally guarded statement. The other option still has a constraint but no transitions. This is necessary because the simulator picks any viable option nondeterministically. So if there is no constraint on one option, it can always be chosen, even if the constraint on another option is met.

Loops allow for repetition of expressions. Ruby contains different ways to create loops, our compiler only supports '`loop`', '`while`', '`unless`', '`for`' and '`each`' (partially). Other ways to create loops are using the '`map`' method. '`map`' executes a function on each element of the array returns the resulting array.

$$\mathcal{C}[\![\ \textbf{if}\ \ \mathit{Expr}\ \ \textbf{then}\ \ \mathit{Stmts}\ \mathit{Rest}\ \textbf{end}]\!] \quad \Rightarrow \quad \textbf{choice \{}\ \textbf{o \{}\ \textbf{constraint "}\ \mathcal{C}[\![\mathit{Expr}]\!]\ \textbf{"}\ \mathcal{C}[\![\mathit{Stmts}]\!]\ \textbf{\}}\ \textbf{o \{}\ \textbf{constraint "!(}\ \mathcal{C}[\![\mathit{Expr}]\!]\ \textbf{)"}\ \mathcal{C}[\![\mathit{Rest}]\!]\ \textbf{\}}\ \textbf{\}}$$

$$\mathcal{C}[\![\ \textbf{elsif}\ \ \mathit{Expr}\ \ \textbf{then}\ \ \mathit{Stmts}\ \mathit{Rest}]\!] \quad \Rightarrow \quad \mathcal{C}[\![\ \textbf{if}\ \ \mathit{Expr}\ \ \textbf{then}\ \ \mathit{Stmts}\ \mathit{Rest}\ \textbf{end}]\!]$$

$$\mathcal{C}[\![\ \textbf{else}\ \ \mathit{Stmts}]\!] \quad \Rightarrow \quad \mathcal{C}[\![\mathit{Stmts}]\!]$$

Figure 3.22: Compilation schemes for '`if/elsif/else`' expressions.

$$\mathcal{C}[\![\ \textbf{if}\ \ \mathit{Expr}\ \ \textbf{then}\ \ \mathit{Stmts}\ \mathit{Rest}\ \textbf{end}]\!] \quad \Rightarrow \quad \textbf{choice \{}\ \textbf{o \{}\ \textbf{constraint "}\ \mathcal{C}[\![\mathit{Expr}]\!]\ \textbf{"}\ \mathcal{C}[\![\mathit{Stmts}]\!]\ \textbf{\}}\ \mathcal{C}[\![\mathit{Rest}]\!]\ [\![\mathcal{C}[\![\mathit{Expr}]\!]\ ]\!]\ \textbf{\}}$$

$$\mathcal{C}[\![\ \textbf{elsif}\ \ \mathit{Expr}\ \ \textbf{then}\ \ \mathit{Stmts}\ \mathit{Rest}]\!]\ [\![\mathit{constraints}]\!] \quad \Rightarrow \quad \textbf{o \{}\ \textbf{constraint "}\ \mathcal{C}[\![\mathit{Expr}]\!]\ \textbf{\&\& !(}\ \mathit{constraints}\ \textbf{) "}\ \mathcal{C}[\![\mathit{Stmts}]\!]\ \textbf{\}}\ \mathcal{C}[\![\mathit{Rest}]\!]\ [\![\mathit{constraints}\ \textbf{\&\& (}\ \mathcal{C}[\![\mathit{Expr}]\!]\ \textbf{)}]\!]$$

$$\mathcal{C}[\![\ \textbf{else}\ \ \mathit{Stmts}]\!]\ [\![\mathit{constraints}]\!] \quad \Rightarrow \quad \textbf{o \{}\ \textbf{constraint "!(}\ \mathit{constraints}\ \textbf{) "}\ \mathcal{C}[\![\mathit{Stmts}]\!]\ \textbf{\}}$$

Figure 3.23: Alternative compilation schemes for '`if/elsif/else`' expressions.

'`loop`'s are modelled using the AML '`repeat`' statement (Figure 3.24). This creates a loop in the transition system over the enclosed transitions.

$$\mathcal{C}[\![\ \textbf{loop do}\ \ \mathit{Stmts}\ \ \textbf{end}\ ]\!] \quad \Rightarrow \quad \textbf{repeat \{}\ \mathcal{C}[\![\mathit{Stmts}]\!]\ \textbf{\}}$$

Figure 3.24: Compilation schemes for loops.

'`while`' loops are also modelled using the AML '`repeat`' statement. The compilation scheme is shown in Figure 3.25. The statement body now consists of two '`option`'s. The first option has a constraint identical to the loop condition. The body of the '`while`' loop is placed in this option. The second option has the same constraint, but negated. In this second option, only a '`stop_repetition`' is added to jump out of the loop. To support the '`redo`' jump statement, the code generator adds a state before the repeat statement.

'`for`' loops are similar to the '`each`' method on arrays. For this reason, the compiler rewrites the '`each`' method to a '`for`' loop prior to compiling it to AML.

To understand how to compile the loop, we need to understand the difference with the '`while`' loop. The major difference is that the '`for`' loop implicitly keeps track of the current element in the array. This makes it possible to use the '`next`' statement inside a '`for`' loop. The '`next`' statement skips the rest of the loop and starts it again with the next element of the array. The compilation steps are

$\mathcal{C}[\![$ **while** *Expr* **do** *Stmts* **end** $]\!]$ $\Rightarrow$

```
state "redo"
repeat {
  o {  constraint " C[[Expr]] "
    C[[Stmts]]
  }
  o {  constraint " !( C[[Expr]] ) "
    stop_repetition
  }
}
```

$\mathcal{C}[\![$ **redo** $]\!]$ $\Rightarrow$ **goto "before_loop"**

$\mathcal{C}[\![$ **break** $]\!]$ $\Rightarrow$ **stop_repetition**

Figure 3.25: Compilation schemes for '`while`' loops and the '`redo`' and '`break`' keywords.

$\mathcal{C}[\![$ **for** *variable* **in** *Expr* **do** *Stmts* **end** *Rest*$]\!]$ $\Rightarrow$

```
D[[variable = Expr [ 0 ] ]]
 var counter , :integer , 0
C[[Rest]]
 state "next"
 repeat {
   o {  constraint " Cond "
     C[[variable = Expr[counter]]]
     C[[counter = counter + 1]]
      state "redo"
     C[[Stmts]]
   }
   o {  constraint " !( Cond )"
      stop_repetition
   }
 }
```

Where $Cond = counter < length(Expr)$

$\mathcal{C}[\![$ **next** $]\!]$ $\Rightarrow$ **goto "next"**

Figure 3.26: Compilation schemes for '`for`' loops and the '`next`' keyword.

shown in Figure 3.26.

First we need to be able to keep track of the current element. To do so we create a variable called '`counter`'. This counter keeps track of the position in the array we are currently at. At the beginning of every iteration, we first check if we are at the end of the array and if so, we stop the iteration. If we are not at the end, we can get the item at the current position and increment our counter for the next iteration. After these two updates, we add a new state. We go back to this state whenever we want to '`redo`' the current iteration. If we want to jump to the next iteration (by using '`next`'), we can go to the '`next`' state.

Functions definitions are modelled as if the function is a system under test (SUT). The compilation schemes are shown in Figure 3.27. Each compiled function consists of a process with an internal channel to receive arguments and to send return values. An internal channel is used for communication between symbolic transition systems (STS's). The parameters of a function are label parameters on the stimulus, but also state variables in the process.

A function call is simulated by sending this stimulus to the model. Upon receiving this stimulus, the model updates the corresponding state variables with the values of the label parameter. Function return is simulated by receiving a response from the function process. The model constrains the label parameters on this transition to the return expression.

While Ruby supports implicit returns (i.e. any last expression in the function is a return expression), our compiler only supports explicit returns.

$$\mathcal{C}\left[\!\left[\begin{array}{l}\textbf{def } foo\ (\ variables\ ) \\ \quad Stmts \\ \textbf{end}\end{array}\right]\!\right] \Rightarrow$$

internal "foo**_channel "**
process(" foo **") {**
  stimulus "call", "_arguments" $\Rightarrow$ **{** $\mathcal{A}[\![variables]\!]$ **},**
    on: "foo**_channel"**
  response "return", "_return" $\Rightarrow$ $\langle foo, \Gamma\rangle_{output}$ **,**
    on: "foo**_channel"**

  $\mathcal{B}[\![variables]\!]$

  receive "call",
    constraint: "_arguments == _arguments",
    update: " $\mathcal{E}[\![variables]\!]$ **"**

  $\mathcal{C}[\![Stmts]\!]$
  state "exit"
}

$\mathcal{A}[\![variable : variables]\!] \Rightarrow \mathcal{A}[\![variable]\!]$ **,** $\mathcal{A}[\![variables]\!]$

$\mathcal{A}[\![variable]\!] \Rightarrow variable \Rightarrow \Gamma(variable)$

$\mathcal{B}[\![variable : variables]\!] \Rightarrow \mathcal{B}[\![variable]\!]$
$\mathcal{B}[\![variables]\!]$

$\mathcal{B}[\![variable]\!] \Rightarrow$ **var** $variable$ **,** $\Gamma(variable)$

$\mathcal{E}[\![variable : variables]\!] \Rightarrow \mathcal{E}[\![variable]\!]$ **;** $\mathcal{E}[\![variables]\!]$

$\mathcal{E}[\![variable]\!] \Rightarrow variable$ **= _arguments.** $variable$

$$\mathcal{C}\left[\!\left[\begin{array}{l}\textbf{def } foo\ () \\ \quad Stmts \\ \textbf{end}\end{array}\right]\!\right] \Rightarrow$$

internal "foo**_channel "**
process(" foo **") {**
  stimulus "call", on: "foo**_channel"**
  response "return", "_return" $\Rightarrow$ $\Gamma_{output}(foo)$ **,**
    on: "foo**_channel"**

  receive "call"

  $\mathcal{C}[\![Stmts]\!]$
  state "exit"
}

$\mathcal{C}[\![Stmt : Stmts]\!] \Rightarrow \mathcal{C}[\![Stmt]\!]$
$\mathcal{C}[\![Stmts]\!]$

$\mathcal{C}[\![$ **return** $Expr]\!] \Rightarrow$ **send "function_return",**
  constraint: " **_return ==** $\mathcal{C}[\![Expr]\!]$ **"**
goto "exit"

Figure 3.27: Compilation schemes for function defintions and '`return`' statements.

Objects can be expressed as a set of properties, or state, and shared code between objects of the same type that use and manipulate the state[12].

A minimal representation of an object in AML could be a tuple of primitives to represent the state and an AML process for each method of the object.

These models are similar to the compiled Ruby functions by our compiler. They first receive a stimulus containing the argument of the method and lastly send a response with the return value.

For the method to read from and write to the state of its object, we need to add the state tuple as a label parameter for the stimulus.

In case the method alters the state of the object, we must add the state as a label parameter to the response. This allows for the environment where the method was called to update its object. Most methods also have a return value. To return two values, we can create another tuple. This tuple contains both the updated state and the return value. The calling environment can then extract the return value from it when needed.

Figure 3.28 shows an example of objects in AML. In the example we have a user object that contains a '`name`' and '`last_name`' property and a '`full_name`' method that accepts a single argument '`reverse`'.

- On lines 3 to 5 and 7 to 9, we define a Ruby variable for our user's state and return tuple.
- On lines 14 to 18, we define the labels for calling to and receiving from the '`full_name`' method.
- On line 23, we create the initial state for our user John, which we then send to the method as the first argument on line 24.
- On line 27 to 29, we refresh our old state with the newly received state and use the return value to store John's last name in a variable.
- Lines 32 to 62 are the STS representation of our '`full_name`' method. In Ruby this would be a function definition and as such the model is similar to a model as generated by our compiler.

```
internal 'method_full_name'

def userState
  { 'name' => :string, 'last_name' => :string }
end

def fullNameReturn
  { 'state' => userState, 'value' => :string }
end

process('user') {
  response 'function_call', '_arguments' => {
    'self' => userState,
    'reverse' => :boolean
  }, on: 'method_full_name'
  stimulus 'function_return', '_return' => fullNameReturn, on: 'method_full_name'

  var 'john', userState
  var 'johns_full_name', :string

  update "john = {'name' => 'John', 'last_name' => 'Doe', '_last_return' => ''}"
  send 'function_call',
       constraint: "_arguments = {'self' => john, 'reverse' => true}"

  receive 'function_return',
          constraint: '_return == _return',
          update: 'john = _return.state; johns_full_name= _return.value'
}

process('full_name') {
  stimulus 'function_call', '_arguments' => {
    'self' => userState, 'reverse' => :boolean
  }, on: 'method_full_name'
  response 'function_return', '_return' => {
    'state' => userState, 'value' => :string
  }, on: 'method_full_name'

  var 'self', userState
  var 'reverse', :boolean
  var 'full_name', :string

  receive 'function_call',
          constraint: '_arguments == _arguments',
          update: 'self = _arguments.self; reverse = _arguments.reverse'

  choice {
    o {
      constraint 'reverse == true'
      update "full_name = self.last_name + ', ' + self.name"
    }
    o {
      constraint '!(reverse == true)'
      update "full_name = self.name + ' ' +  self.last_name"
    }
  }

  send 'function_return',
       constraint: "_return == {'state' => self, 'value' => full_name}"
}
```

Figure 3.28: Potential approach to using objects in AML.

Chapter 4

Evaluation

In this section we show how we use an empirical testing method to evaluate our compiler and discuss how testers can use the compiler in practice.

4.1 Testing

Since we have not given a proof of the semantic equivalence of Ruby functions and the AML into which they are compiled (due to time constraints), we provide a procedure to verify their equivalence through exhaustive testing. Validation of the transformations is done using the equivalence property of the generated AML model. This property states that for a given input, the output for both the Ruby function and the generated AML model are equal.

Using Axini's simulator, we can send stimuli to our model and check if it can send a response with the return value of our Ruby function. Thus, the evaluation process boils down to the following steps:

1. Compile the input, the Ruby function definition, to a AML model.
2. Evaluate the input in Ruby to create a callable Ruby function.
3. Call the Ruby function with a specific, manually determined, input and record the output.
4. Send a stimulus to the AML model with the same specific input as its parameter.
5. Check if the model specifies that the Ruby function can send a response with the previously recorded output as its parameter.

Step 5 yields a Boolean value, which is true when the model can send a response with the Ruby output as its parameter. This means that the constraints on this transition can be satisfied and the output of the AML model is equal to the output of the original Ruby function. If Step 5 returns false, then the model was not expected to return a response or the constraints could not be satisfied.

This method can be used to exhaustively test if all paths through the original function and the generated model are covered. If a path is not covered, we cannot say anything about the validity of the compilation of the Ruby function for values that lead to that path.

After verifying the correctness of a sufficiently varied amount of compiled functions, we may gain some trust that the compilation retains the function's semantics. This would make it unnecessary to exhaustively test every compiled instance.

Alternatively, we can automate this test process by using the simulator to generate input values to cover every trace through the symbolic transition system (STS). By recording the response send by the modelled function, we can compare the output of the Ruby function for each generated input value. However, this approach does not ensure that the test covers every path through the original Ruby function. To evaluate the coverage in the original function, a Ruby library such as *SimpleCov*[1] can be used. This automated approach has the benefit that it can be added as the final step in our compilation pipeline. A successful test then immediately assures that the compilation was successful.

[1] `https://github.com/colszowka/simplecov`

4.2 Usability

To use the generated models in place of AML functions, we want them to be (partially) solvable. This means that we can generate input that will satisfy all constraints for a certain trace.

However, current test strategies traverse the STS top to bottom, one transition at a time and solve constraints on a single transition directly. This means that it could generate input that will satisfy the first constraint, but not the constraints on any of the following transitions. Because the compiled function is inlined in place of the function call (see Figure 4.1), we need to solve for at least one trace through the entire model of the compiled function.

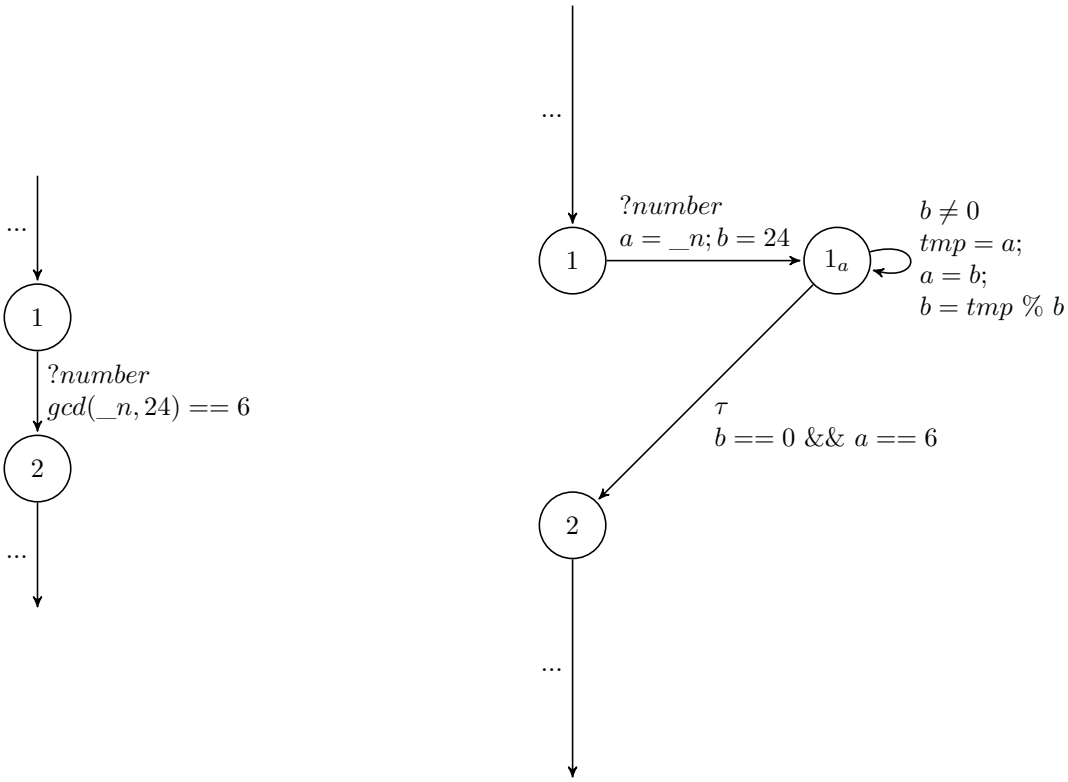

Figure 4.1: The constrained transition (left) is replaced with the compiled *gcd* function (right).

Prior research shows it is possible to create a test strategy for a specific trace [16]. While this is an exponential time algorithm, it is only exponential in the number of observable transitions. Our generated models will only have traces that start with a single stimulus and then directly end with a single response. Therefore the trace coverage strategy is applicable to our generated models.

Chapter 5

Related work

There exists prior research on modelling software as a mathematical model and creating an executable semantics for a language by different authors [18, 19, 22]. In this section we will briefly introduce their work and discuss why ours is still relevant.

5.1 Model extraction

C model extraction. In 1999, Holzmann and Smith researched automated model extraction for C to provide a faster, more reliable method of formal software verification. The design of software under development is subject to frequent change. Formal verification methods of that time required great amounts of time and expertise, making them costly. Their motivation was that formal verification of software during its development is not feasible. By automating the modelling process, it would become significantly faster and thus applicable during development. [19]

Their approach is to extract the control-flow skeleton and fill it with the message-passing operations. By using a look-up table, they convert all the C statements into Promela statements. They argue that the rest of the C statements can be abstracted similarly. However, they do not provide any examples.

The main difference with our research is the use of the extracted model. Our generated models are used for test generation and execution on the system under test (SUT). Holzmann and Smith's generated models are used for model checking the source code.

Java model extraction. In 2000, Havelund and Pressburger researched how to do automated model extraction for Java to provide verification and debugging of Java programs (*Java Pathfinder*). Their motivation was to apply formal software verification during development without consulting experts. [18]

Brat et al. expanded *Java Pathfinder* to extract models from Java byte-code [7]. This allows programs that do not include the source of external libraries to be model checked. Anand et al. added symbolic execution [1]. This makes the model checking of programs that accept inputs from unbounded domains possible. Currently, the symbolic execution extension only handles numeric inputs.

The main difference with our research is the use of the extracted model. Our compiler is used to generate parts of models to generate and execute tests on other systems. *Java Pathfinder* was created to apply model checking to Java programs during development.

5.2 Executable semantics

Executable semantics for Python. In 2009, Smeding researched an executable semantics for a subset of Python to provide a more legible reference implementation. Because there exists no formal semantics for Python, new implementations are often derived from CPython, the de facto implementation. This implementation is not written with legibility as its primary focus.

Smeding provided an operational semantics for a subset of Python and created an interpreter based on the same sources. By reverse engineering the semantics of CPython, he wrote a test suite and ran it across various compilers to test the accuracy of their interpreter.

While our research has shown that the symbolic transition system (STS) can be used an executional model for Ruby, our main focus was not on defining an executable semantics. Smeding's work did not use an underlying formalism used in model-based testing (MBT). As such, his work is not directly applicable in the area of MBT.

Chapter 6

Conclusion

6.1 Summary

In this thesis, we have shown a proof of concept compiler to compile Ruby functions into symbolic transition systems (STS's) using AML as the modelling language. Using this compiler, we can use a subset of Ruby to facilitate input generation and trace coverage for complex computations in AML.

The implementation of the compiler consists of five components. First, the parser parses Ruby and outputs an abstract syntax tree (AST). Secondly, the checker ensures that the input consists of only Ruby source code from the supported subset. Then the rewriter desugars the input. After desugaring, the propagator creates a record of all the variables and their types. This record is required as AML is statically typed and we have to be certain variables do not change types. Finally, the compiler transforms the Ruby input into an AML model.

We also discussed how we tested our implementation by exhaustively comparing input/output pairs between the Ruby function definition and the generated AML model. Then we discussed how our compiler could be used in practice. By using prior research on solving traces, the Ruby function can be solved to acquire the input for the desired output. Unfortunately, we were not able to apply our work to an industry model due to time constraints.

6.2 Research Questions

The answer to the first subquestion, *What relations between Ruby and STS's can we identify that help us in making such translations?*, follows from the compilation pipeline in Chapter 3 (Ruby-AML Compiler). Section 3.6 (Code Generator) shows the compilation schemes that map Ruby to AML. Because most AML expressions directly represent partial STS's, it also maps Ruby to an equivalent STS. This section contains relatively simple rules that show that some Ruby and AML expressions have nearly identical semantics. Examples are AML state variables to represent Ruby local variables, the AML '`choice`' statement with mutually exclusive constraints to represent Ruby '`if/else`', AML '`repeat`' statements to represent Ruby loops and label parameters are used for message passing between objects. An AML model can thus be seen as an executional model for Ruby.

The answer to the second research question, *For what subset of Ruby can we perform such translations?*, can also be found in Chapter 3 (Ruby-AML Compiler). In short, the following concepts are supported by our proof of concept: static variable assignment (including operator assignment), arithmetic expressions, Boolean expressions and control flow expressions (`if/elsif/else, unless, while, until, for, break, redo, next`. In Section 3.6 we mention how we implemented these concepts and we discuss the theory behind implementing objects with state and methods.

The third research question, *Can the constraints in the resulting STS's be solved by the finite domain solver such that we can indeed generate inputs that will cover all possible traces through the function?* is answered in Chapter 4 (Evaluation). By using research on trace coverage [16], we concluded that if the generated model does not use AML functions, traces from the beginning to the end of the model can be solved. Because the trace coverage algorithm is exponential in the number of observable transitions, our

generated models are suitable for the algorithm considering they only have two observable transitions (input arguments and return value) per trace.

By answering these three subquestions, we can answer our main research question:

> Is it possible to translate a Ruby function into an STS representation that effectively implements the same function, such that we can then use constraint solving to generate inputs for the function that will cover all traces through the function?

Yes, it is possible to translate a Ruby function in this way, provided only the supported subset of Ruby is used.

6.3 Future work

The soundness of the models generated by our compiler relies on the correctness of the transformations. By formulating a semantic proof of the transformations, we can retain the mathematical basis of the STS formalism. Only then can we rely on the results of test cases generated from the models.

Additionally, model checking can now be done on source code that adheres to the subset. By compiling the source code to an STS, we can use an STS model checker to verify the generated model. Assuming the transformations are sound, the model checker is effectively checking the source code. However, such a model checker does currently not exist in the literature.

Furthermore, our compiler only supports a subset of the Ruby language. The subset should be expanded to apply model checking in development. Interesting concepts to include are regular expressions and objects (including classes and object initialisation) as they are commonly used throughout Ruby projects.

Bibliography

[1] S. Anand, C. S. Pasareanu, and W. Visser. JPF-SE: A symbolic execution extension to Java PathFinder. In *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings*, pages 134–138, 2007.

[2] B. Beizer. *Software testing techniques (2. ed.)*. Van Nostrand Reinhold, 1990. ISBN 978-0-442-20672-7.

[3] B. Beizer. *Black-box testing - techniques for functional testing of software and systems*. Wiley, 1995. ISBN 978-0-471-12094-0.

[4] A. Belinfante, L. Frantzen, and C. Schallhart. Tools for test case generation. In *Model-Based Testing of Reactive Systems, Advanced Lectures [The volume is the outcome of a research seminar that was held in Schloss Dagstuhl in January 2004]*, pages 391–438, 2004.

[5] A. Bertolino. Software testing research: Achievements, challenges, dreams. In *International Conference on Software Engineering, ISCE 2007, Workshop on the Future of Software Engineering, FOSE 2007, May 23-25, 2007, Minneapolis, MN, USA*, pages 85–103, 2007.

[6] H. M. van der Bijl. *On changing models in Model-Based Testing*. PhD thesis, 5 2011. IPA Dissertation Series No. 2011-07.

[7] G. Brat, K. Havelund, S. Park, and W. Visser. Java pathfinder-second generation of a Java model checker. In *In Proceedings of the Workshop on Advances in Verification*, 2000.

[8] M. Bravenboer and E. Visser. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28, 2004, Vancouver, BC, Canada*, pages 365–383, 2004.

[9] V. de Bruijn. Model-based testing with graph grammars. Master's thesis, Universiteit Twente, the Netherlands, 2013.

[10] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Trans. Software Eng.*, 4(3):178–187, 1978.

[11] R. De Nicola and M. Hennessy. Testing equivalences for processes. *Theor. Comput. Sci.*, 34: 83–133, 1984.

[12] K. Dickey. Scheming with objects. *AI Expert*, 7(10):24–33, 1992.

[13] L. Frantzen. *STSimulator: A Library to Simulate Symbolic Transition Systems*, 2007. Version 271007.

[14] L. Frantzen, J. Tretmans, and T. A. C. Willemse. Test generation based on symbolic specifications. In *Formal Approaches to Software Testing, 4th International Workshop, FATES 2004, Linz, Austria, September 21, 2004, Revised Selected Papers*, pages 1–15, 2004.

[15] L. Frantzen, J. Tretmans, and T. A. C. Willemse. A symbolic framework for model-based testing. In *Formal Approaches to Software Testing and Runtime Verification, First Combined International Workshops, FATES 2006 and RV 2006, Seattle, WA, USA, August 15-16, 2006, Revised Selected Papers*, pages 40–54, 2006.

[16] A. Ghaffari. Trace coverage strategy for symbolic transition system. Master's thesis, Universiteit van Amsterdam, the Netherlands, 2016.

[17] A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of javascript. *CoRR*, abs/1510.00925, 2015.

[18] K. Havelund and T. Pressburger. Model checking Java programs using Java pathfinder. *Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[19] G. J. Holzmann and M. H. Smith. Software model checking. In *Formal Methods for Protocol Engineering and Distributed Systems, FORTE XII / PSTV XIX'99, IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XII) and Protocol Specification, Testing and Verification (PSTV XIX), October 5-8, 1999, Beijing, China*, pages 481–497, 1999.

[20] E. F. Moore. Gedanken-experiments on sequential machines. volume 34 of *Annals of Mathematics Studies*, pages 129–153. Princeton University Press, 1956.

[21] J. G. Politz, A. Martinez, M. Milano, S. Warren, D. Patterson, J. Li, A. Chitipothu, and S. Krishnamurthi. Python: the full monty. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 217–232, 2013.

[22] G. J. Smeding. An executable operational semantics for python. Master's thesis, Universiteit Utrecht, the Netherlands, 2009.

[23] O. Taipale, J. Kasurinen, K. Karhu, and K. Smolander. Trade-off between automated and manual software testing. *Int. J. Systems Assurance Engineering and Management*, 2(2):114–125, 2011.

[24] G. Tassey. The economic impacts of inadequate infrastructure for software testing. Technical report, National Institute of Standards & Technology, 2002.

[25] M. Timmer, E. Brinksma, and M. Stoelinga. Model-based testing. In *Software and Systems Safety - Specification and Verification*, pages 1–32. 2011.

[26] J. Tretmans. Model based testing with labelled transition systems. In *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*, pages 1–38, 2008.

Appendix A

Checker Errors

Unsupported node. When it encounters any type of node that it does not recognise, it will return an error with the message "Unsupported node". An example of this is the '`yield`' keyword, which will return this error.

Unexpected jump. When it encounters one of the jump keywords ('`redo`', '`next`', '`break`') outside a loop, it will return this error.

Unsupported nested function definition. When it encounters a function definition that is not the root of the AST, it will return this error.

Unsupported argument type. When it encounters any argument that is not a required argument, it will return this error.

Unsupported splat in right hand side. On mass assignment, it is possible to add the splat operator on the left and right hand side. Currently, the compiler does not support this operator on the right side. Example: '`a, b, c = 1, *[2,3]`'.

Unsupported function call. The compiler only supports a handful of operators and functions that can be either rewritten in Ruby or transformed directly to the Axini Modelling Language (AML). When the checker encounters a function call to a function that does not belong to this set, it will return error.

Unsupported recursion. Only tail recursion is supported. The compiler supports recursion, but only when it is part of a return statement. When the checker detects a recursive call at any other place, it will add this error.

Unexpected list type, expected variable. While the compiler supports enumeration over an Array using 'for' and 'each', it only does so when the Array is a variable.

Parallel assignment is serialized. From the programmer's perspective, mass assignment in Ruby looks like a parallel assignment. This means that swapping variables can be done on a single line like so: '`a, b = b, a`'. Our compiler only serialises the assignment. As a result, '`b`' would be assigned its own value because it was assigned to '`a`' before ('`a = b; b = a` $\Rightarrow$ `b = b`'. When the input contains a mass assignment with variables on the right hand side that also occur on the left hand side, this error is returned.

Appendix B

Axini Modelling Language

Due to the confidential nature of the information in this appendix, it has been removed in this publication. For more information about the Axini Modelling Language (AML), please contact Axini B.V. (info@axini.com).