Test-inspired runtime verification

Using a unit test-like specification syntax for runtime verification

ADAM RENBERG

Master's Thesis at CSC
Supervisor Valtech: Erland Ranvinge
Supervisor CSC: Dr. Narges Khakpour
Examiner: Prof. Johan Håstad

Abstract

Computer software is growing ever more complex, and more sophisticated tools are required to make sure the software operates in a correct way — i.e. according to its specification. Traditional approaches to program verification have much to give, but they also have their disadvantages. While formal methods can give nice mathematical proofs about the correctness of programs, they suffer from complexity and are difficult to use. Often they work only on a constructed system model, not the actual program. Testing, on the other hand, has a simple syntax and great tool support, and it is in widespread use. But it is informal and incomplete, only testing the specific test cases that the test-writers can come up with.

A relatively new approach called *runtime verification* is an attempt for a lightweight alternative. It verifies a program's actual *execution* against its specification, possibly while the program is running.

This work investigates how testing, and specifically unit testing, can be combined with runtime verification. It shows how the syntax of unit tests, written in the target program's programming language, can be used to inspire the syntax for specifications for runtime verification. Both informal and formal specifications are described and supported.

A proof-of-concept framework for Python called *pythonrv* is implemented and described, and it is tested on a real-life application. A formal foundation is constructed for specifications written in a subset of Python, enabling formal verification. Informal specifications are also supported, with full power of Python as specification language.

The result shows that the proof-of-concept framework allow for effective use of runtime verification. It is easy to integrate into existing programs, and the informal specification syntax is quite simple. Many interesting properties can be verified using it that ordinary tests would have trouble with.

It also shows that formal specifications can be written in Python, but in a more unwieldy syntax and structure.

The recommendation for future work lies in improving the specification syntax, using unit testing concepts such as expectations, and on working to make the formal specification syntax more like that of its informal sibling.

Referat

Test-inspirerad runtime-verifiering

Mjukvara växer sig allt mer komplexa, och mer sofistikerade verktyg krävs för att säkerställa att programmen fungerar korrekt — at den opererar enligt sin specifikation. Traditionella tillvägagångssätt till programverifiering har mycket att ge, men de har också sina nackdelar. Medan formella metoder kan ge trevliga matematiska bevis om korrektheten av program, lider de av komplexitet och är svåra att använda. Ofta opererar de bara på en konstruerad systemmodell, inte det faktiska programmet. Testning, å andra sidan, har en enkel syntax och bra verktygsstöd, och är i utbredd användning. Men det är informellt och ofullständigt, testar bara de specifika testfall som testkrivarna kan komma på.

En relativt ny metod kallad runtime-verifiering är ett försök för ett lättviktigt alternativ som verifierar ett programs faktiska *exekvering* mot dess specifikation, eventuellt medan programmet kör.

Det här arbetet undersöker hur testning, och specifikt enhetstestning, kan kombineras med runtime-verifiering. Det visar hur syntaxen för enhetstester, skrivna i programmets programmeringsspråk, kan användas som inspiration för specifikationer för runtime-verifiering.

En proof-of-concept-implementation för Python kallad *pythonrv* implementeras och beskrivs, och testas på en verklig applikation. En formell grund framställs för specifikationer skrivna i en delmängd av Python, vilket möjliggör formell verifiering. Informella specifikationer stöds också, med hela kraften av Python som specifikationsspråk.

Resultaten visar att proof-of-concept-implemenationen tillåter för effektiv användning av runtime-verifiering. Den är enkel att integrera i existerande program, och den informella specifikationssyntaxen är hyfsat enkel. Många intressanta egenskaper kan verifieras med den som vanliga tester skulle ha problem med.

De visar också att formella specifikationer kan skrivas i Python, men i en mer omständigt syntax och struktur.

Rekommendationen för framtida arbete är att förbättra specifikationssyntaxen, genom att använda koncept från enhetstestning såsom förväntningar, och genom att göra den formella specifikationssyntaxen mer som den av sin informella kusin.

Preface

This is a degree project in Computer Science at the Royal Institute of Technology (KTH), Stockholm. The work was done at Valtech Sweden, an IT consultancy.

I've been fortunate to have had two great supervisors: Erland Ranvinge at Valtech and Dr. Narges Khakpour at the School of Computer Science and Communication, KTH. They have been of great help, both in the conception of the initial idea, and during the project, discussing problems and giving feedback on the work and this report. Thank you!

I'd also like to thank Valtech for the opportunity to do this degree project as part of a trainee program, and for all the great colleagues there giving their feedback and support, especially on the development of the internal web application described in Chapter 6.

And last, I'd like to thank the many proofreaders. Without them, this report would be a lot worse. Any errors still in the report are mine and mine alone.[1]

---

[1]For an interesting take on this last sentence, see *The Preface Paradox* [1, 2].

Contents

Chapter 1

Introduction

Due to the increasing size and complexity of computer software it has become increasingly difficult, if not impossible, to assure oneself that the software works as desired. This is where verification can be helpful. Of the various approaches used for verification, *testing*, in all its forms, is the one familiar to most developers, and in it is widespread use. The introduction of agile development practices and test-driven development has also popularized the concept of *unit testing*, a form of testing in which small modules of a program or system are tested individually.

While testing is popular and often works well, it is incomplete and informal, and thus yields no proof that the program does what it should — i.e. follows its specification. Formal verification techniques, such as theorem proving and *model checking* (and its bounded variant), can give such proofs. However, they suffer from complexity problems (such as *incompleteness*, *undecidability*) and practical issues, such as the so-called *state explosion* problem.

A relatively new approach in this area is *runtime verification*, in which the program *execution* is verified against its specification, at runtime. With the specification written in a suitably formal language, the execution can be given a mathematical proof that it follows the program's specification.

This is where this report takes off, with formal verification on one side and testing on the other, and with runtime verification somewhere in between. Could the syntax and tooling of unit testing be used to improve the ease of using runtime verification?

1.1 Problem Statement

How can runtime verification specifications be written in a manner that uses the syntax of the target program's programming language, and resembles the structure of unit tests? Can we still give a formal semantics to the specification language, or a part of it? How can we bridge the spectrum of different approaches for verification, creating something in between the formal and informal techniques?

1.2   Motivation

Checking that a program works correctly is of great interest to software developers. Formal verification techniques are helpful, but as mentioned above, traditional methods can be impractical with larger programs, and verification by testing is informal and incomplete. Runtime verification can here be a lightweight addition to the toolbox of verification techniques.

The specification languages used by runtime verification approaches are often based on formal languages/formalisms (e.g. logic or algebra) and not written in the target program's programming language. This means that writing the specifications requires specific knowledge and expertise in mathematics. It also requires mental context-switching, between writing the program and writing the specification, and special tools to support this specialised language's syntax.

In contrast, unit testing frameworks often utilise the programming language to great effect, and they are a common part of the software development process.

However, formal specifications allow for mathematical reasoning and proofs about properties of the program. A combination of the simpleness of the syntax and tool-support for testing with the mathematical properties of formal specifications could add more certainty to the verification in software development.

If runtime verification specifications more resembled unit tests, and were written in the target program's programming language, it might popularise the use of runtime verification for checking the correctness of programs.

1.3   Structure of this Report

The rest of this report is structured as follows. This chapter gives an introduction to the report and the problem statement. Chapter 2 gives a background to the subject of verifying program correctness and an overview of formal methods, testing and runtime verification. Chapter 3 continues by describing the previous research on runtime verification, the syntax of specification languages and different approaches to code instrumentation. Chapter 4 gives an overview of the current ideas in unit testing.

Chapter 5 describes the approach this work takes to solving the problem stated in Section 1.1. It describes the syntax, instrumentation and verification techniques used in a proof-of-concept implementation, and gives a formal foundation to a subset of the syntax. Chapter 6 then gives an evaluation of applying this work on a real-life web application.

Conclusions and a discussion of this and future work is done in Chapter 7.

The first time an important concept is introduced it is written in *italics*. If such a concept is not explained in the text, it has a brief description in the Glossary, see Appendix A.

1.4 Limitations

Naturally, there are some areas that this report does not attempt to solve. The report describes a proof-of-concept implementation of a runtime verification framework for Python, with specifications written as Python code. It discusses both formal and informal specifications, and shows how verification can be done for both. It does not attempt to formalize Python. It gives a formalization for only a subset of Python — for specifications constructed in a special way.

Also, it only deals with a single thread of execution — no multi-threading or multiple processes. To limit the scope of the framework, it doesn't directly expose real-time data to the specifications.

No performance testing or benchmarking is done on the implementation.

Chapter 2

Background

Runtime verification is a new area of research, but the research on other forms of verification, such as formal methods and testing, goes back several decades. The terms used in this area are used slightly differently by different researchers, so this chapter starts by giving definitions for some concepts used in this report. Then we give a short overview of the research areas related to the subject of this report. Section 2.2 explains what is meant by proving the correctness of programs, and Section 2.3 explains what testing is. And finally, Section 2.4 describes runtime verification and its place in the set of verification approaches.

2.1 Terms and Definitions

Verification is the very broad concept of checking that a program does what it is supposed to, which can be expressed as "Are we building the program correctly?". This can be contrasted with *validation*, in which we ask: "Are we building the correct program?" [3]. Validation is concerned with whether the specifications really capture the requirements we want for the system. Verification is concerned with assuring us that the program follows its specification, and it is verification in which this report takes an interest. Verification includes everything from manually running the program and parsing through the output, to automated testing setups, to formal methods.

Both formal methods and testing are thus included in the term verification, and they are both described in this chapter.

An essential concept in verification is that of a *system model*. A system model is a construct that represents and describes the behaviour of a system. It is usually an abstraction of the real system, leaving out the details that are of no interest to the task at hand, and making other modeling aspects more prominent.

We use system models constantly in everyday life when we abstract away the details of how things actually work to a more easy-to-grasp model of how it seems to work — e.g., when driving a car, operating a computer, etc. We often get into trouble when the system model becomes too much simplified, or when it conflicts

with the actual system, because we lose lots of information. A too detailed system model can give rise to complexity and noise, however, so a compromise is desirable.

In formal methods, a system model is a mathematical representation of the system that captures and describes its relevant parts — the parts of the system we wish to prove properties about, and reason with formally. In unit testing, the system model is the unit under test, an isolated slice of the system, with the rest of the actual system ignored or mocked away.

2.2 Formal Methods

Formal methods are verification techniques, based in mathematics, for the specification and verification of systems. There are several approaches, with their respective advantages and disadvantages. *Theorem proving* and *model checking* will be discussed below. They both rely heavily on the concept of a proof of correctness.

A correctness proof is a certificate, based in mathematics and logics, that a program/system/function follows its specifications, i.e. does what it is supposed to do.

Research of interest include the early work on formal methods, e.g. by Hoare [4] and Floyd [5], and work on logics , e.g. linear temporal logic (LTL) by Pnueli [6]. The seminal work done by Hoare, Floyd and Pnueli lay the foundation for many interesting approaches. LTL is one of the common formal languages used for specifications in runtime verification.

Theorem proving, as started by Hoare [4], Floyd [5] and others, is the manual, semi-automated, or (not so often) fully automated process of mathematically proving that the system follows its specification. There are many ways of doing such proofs.

One way is to prove that at all points in the program, given inputs satisfying some pre-conditions, the outputs will satisfy the post-conditions. By formulating post-conditions for the exit point(s) of the program so that they follow the specification, and by linking together the pre-conditions of program points with their preceding program points' post-conditions, we know that correct input data will yield correct results.

This way of proving correctness often yields very good results — a complete and thorough correctness proof for the program. But it is slow, hard to automate completely, and therefore requires much manual work. Wading through large programs thus often becomes impractical.

Model checking is the concept of verifying that a *model* of a system (the *system model*) follows its specification. This requires that both the model and the specification is written in a mathematical formalism. Given this, the task becomes to see if the model satisfies the logical formula of the specification. It is often simpler than theorem proving, and can be automated.

The model of the system is usually structured as a finite state machine (FSM), and verification means visiting all accessible states, checking that they follow the

specification (which also can be represented as an FSM). This can be problematic, especially when the state space becomes very big, something known as the *state explosion problem*. There are approaches to address this issue, such as *bounded model checking*, or by using higher-level abstractions.

Proving that a model of a system is correct can be very useful, but it suffers from the inherent flaw of only verifying the model, not the actual system. The model can be difficult to construct, or deviate too far from the system. It can not take the dynamic properties and configuration of the executing code into account.

Runtime verification (Section 2.4) attempts to solve this by dealing directly with the system, creating its model at runtime.

2.3   Testing

Opposite formal methods on the verification spectrum is *testing*, which is the practical approach of checking that the program, given a certain input, produces the correct/acceptable output. Testing consists of a set, or *suite*, of *test cases*. A test case tests some aspect of the program, from large ("It should be possible to go through the shop flow and buy a product") to small ("This function should throw an exception when given a null argument").

Testing can be either manual or *automated*, or something in between. In manual testing a programmer, tester or other stakeholder, runs the program, entering some inputs and inspecting the outputs. Test cases could be documents describing what should happen for given inputs, where to click or type to cause the desired outcome. In *automated* testing the test cases are machine-runnable, so that a computer can run them during the development process, or when checking code into a central repository, or during releases, etc.

Testing is not complete (for all but the most trivial programs, it is impossible to write complete tests), and lacks a formal foundation, so it cannot be used for formal verification. Testing is popular and is used on virtually all development projects, taking up a large part of development time (see e.g. [7]).

Unit testing is the concept of writing tests for small units in a program, such as functions, classes, etc. These tests are used during development to test the functionality of the units. The aim is to reduce the risk of breaking existing functionality when developing new features, or modifying existing code, by preventing regression. Unit testing is described in more detail in Chapter 4.

2.4   Runtime Verification

Runtime verification (RV) is a dynamic approach to checking program correctness, in contrast to testing and the more traditional formal static analysis techniques discussed in Section 2.2. Runtime verification aspires to be a light-weight formal verification technique, see e.g. [8, 9]. It verifies whether properties of a specification hold *during the execution* of a program.

Runtime verification can be both formal, if the specifications are given a formal semantics, or informal, if they are more like tests or other program code.

The specification that should be verified is often written in a formal language, a logic or a calculus, such as LTL [6] (this report shows one exception — see Chapter 5). To build a system model for verifying the properties of the specification, the target program needs to emit or expose certain events and data. The collected events and data are used to build the system model. RV frameworks typically use *code instrumentation* to generate *monitors* for this end.

A monitor is either just part of a recording layer added to the program, which stores the events and data needed for verification, or also the part of the machinery that performs verification.

There are two types of verification: *online* and *offline*. In online verification, the analysis and verification is done during the execution, in a synchronous manner with the observed system. In offline verification, a log of events is analysed at a later time. Online verification allows actions to be taken immediately when violations against the specifications are detected, but with considerable performance cost. Offline verification only impacts the performance by collecting data.

When a violation of the specification occurs, simple actions can be taken (e.g. crash the program, log the error, send emails, etc.), or more complex responses initiated, resulting in a *self-healing* or *self-adapting* system (see e.g. [10]).

Leucker et al. present a definition of RV in [8], together with an exposition of the advantages and disadvantages, similarities and differences, with other verification approaches. In [9], Delgado et al. classify and review several different approaches to and frameworks for runtime verification.

The next chapter examines RV in more detail.

Chapter 3

Introduction to Runtime Verification

As we saw in Section 2.4, runtime verification is a technique for verifying a program's compliance against a specification during runtime. These specifications need to be written somehow, which will be discussed in Section 3.1. Approaches for verification are discussed in Section 3.3. For verification to work, during runtime, the program usually needs to be instrumented in such a way that the verification process can access all pertinent data. This is discussed in Section 3.4

3.1   Specifications

A specification is essentially something that describes the correct behaviour of a system. Specifications come in many forms, from the informal ones like "I want it have cool buttons", to the contractual ones written between companies and their clients, to tests (see e.g. Chapter 4), and to formal specifications, written in formal languages, specifying properties that should verifiably hold for the program (see e.g. Section 3.2.1). It is these last two types of specifications that we are interested in here, and which play an important role in runtime verification.

In general, specifications should be abstract, written in a high-level language, and succinctly capture the desired property. Writing erroneous specifications is of course a possibility; specifications need to be easier for humans to verify than the program's implementation. There is little point in having a specification as complex as the program itself, except for as a point of reference. A program can be seen as an all-encompassing, perfect, always-true, specification of itself.

3.2   Related Work

Temporal Rover, EAGLE, MAC Logic

3.2.1 Formalisms for Specifications

There are several common formalisms for writing specifications, and many papers that expand, rephrase and illuminate on them. Although they can be quite different, they share a common origin in the work done by Floyd [5], Hoare [4], and others before them. Floyd wrote of formulas as specifying in/out properties of statements, and chaining these together to form a formal proof for the program. Hoare elaborated on this idea by basing his proofs on a few axioms of the programming language and target computer architecture, and building the proof from there.

Linear Temporal Logic

Linear temporal logic (LTL) was first discussed by Pnueli [6], and has since been popular in many areas dealing with a system model containing a temporal dimension. LTL is simpler than other logics, but expressive enough to describe many problems of interest for verification. This has been affirmed by the diverse use of LTL by many researchers [6].

LTL uses a system model of *infinite execution traces*, or *histories*, of the states of the execution. LTL specifications are formulas that operate on these states. An LTL formula consists of *propositional variables* that work on the domain model of the state (checking variables, inputs, global state, etc.), the normal logical operators such as negation and disjunction, and some temporal operators. The most basic and common temporal operators are $\boldsymbol{X}$ (*next*), and $\boldsymbol{U}$ (*until*). Other operators can be derived from these, such as $\boldsymbol{G}$ (*globally*, or always) and $\boldsymbol{F}$ (*eventually*, or sometime in the future).

An example LTL formula, taken from a list of common specification patterns [11], could be: S precedes P, i.e. if the state P holds sometime, the state S will hold before it. This is shown in Figure 3.1.

$$\boldsymbol{G}\,P \rightarrow (\neg P\,\boldsymbol{U}\,(S \wedge \neg P)$$

Figure 3.1. An example of an LTL formula. This can be read as: Globally, if P holds, then, before P, S held at some point.

Bauer et al. [12] introduce a three-valued boolean semantics for LTL, calling it LTL$_3$, which takes the values (true, false and ?). This logic is arguably more suited for the finite nature of runtime verification, whereas LTL was designed with infinite traces in mind. The semantics of LTL$_3$ reflect the fact that when verifying runtime verification specifications, the result can not only be that the specification is satisfied or violated; it can be inconclusive as well. For satisfied or violated specifications, no further verification is required — we already know the outcome. For inconclusive results, we need to continue with the verification, as, with future events, the result could change into either satisfied or violated.

There is a counterpart to LTL in the real-time setting called Timed Linear Temporal Logic (TLTL). It introduces clocks to make specifications of real-time

properties possible. It is of great interest to runtime verification, but will not be discussed further here. See e.g. [12] for more.

EAGLE and RuleR

Barringer et al. have written EAGLE [13] and RuleR [14] as runtime verification frameworks and formal logics. Inefficiency weaknesses were found in EAGLE, and RuleR was proposed as a more efficient alternative, with a more lower-level logic. They share much of the same basic ideas, however — they are written by mostly the same people.

EAGLE can be seen as a generalization of other logics, supporting both past and future tense predicates. It is a temporal finite trace monitoring rule-based logic, meaning that a specification consists of a set of rules, which operate on finite sequences of states of the system model, each state being part of a program execution.

A simple example of an EAGLE formula is shown in Figure 3.2. It describes the LTL operator G. $\bigcirc$ is the same as the LTL *next* operator X, and **Form** is the type of F, short for formula. The **max** qualifier denotes that a maximal solution should be found for the equation. $\bigcirc$Always(F) is thus defined to be true in the last state of a given trace. A corresponding **min** qualifier also exist.

$$\textbf{max } \text{Always}(\textbf{Form } F) = F \wedge \bigcirc\text{Always}(F)$$

Figure 3.2. A simple example of an EAGLE formula, semantically describing the LTL G operator. Taken from [13].

The same property can be described in RuleR. If we let $\{r\}$ be the initial set of rule activations, the formula in Figure 3.3 say that rule r requires an observation of r in this state, and that the rule r should hold in the next state.

$$r \; : \rightarrow \; a, r$$

Figure 3.3. A simple example of a RuleR formula, semantically describing the LTL G operator. Taken from [14].

An EAGLE formula describing the LTL formula from Figure 3.1 is shown in Figure 3.4. Note the use of the **min** and **max** quantifiers. Here they mean that PreviouslySometime(F) should be defined as false on the first state, while PreviouslyAlways(F) should be true.

Both EAGLE and RuleR are strictly more powerful than LTL; e.g., they both support specifications that allow matching function calls with function returns, the realm of context-free grammars. LTL does not support this [13, 14].

$$\begin{aligned}
\mathbf{min}\ \text{PreviouslySometime}(\mathbf{Form}\ F) &= F \vee \odot \text{PreviouslySometime}(F) \\
\mathbf{max}\ \text{PreviouslyAlways}(\mathbf{Form}\ F) &= F \wedge \odot \text{PreviouslyAlways}(F)
\end{aligned}$$

$$\text{Always}(P \rightarrow (\text{PreviouslySometime}(S) \wedge \text{PreviouslyAlways}(\neg P))$$

Figure 3.4. A simple example of an EAGLE formula, semantically the same as the LTL formula from Figure 3.1.

Temporal Rover

todo: A short description of Temporal Rover's Logic

Design by Contract

Design by Contract is a verification technique similar to runtime verification, without the ability to describe temporal properties. Design by Contract was introduced by Bertrand Meyer in [15], and has been fully implemented in the Eiffel programming language[1]. An example can be seen in Figure 3.5.

```
put-child (new: NODE) is
  -- Add new to the children of current node
  require
    new /= Void
  do
  ... Insertion algorithm ...
  ensure
    new.parent = Current;
    child-count = old child-count + 1
  end
```

Figure 3.5. An example showing the use of the Design-by-Contract concepts of pre- and post-conditions, written in Eiffel. Note the use of the language construct `old` to get the value of an attribute at before the function executed. Example taken from [15].

A contract is the idea that functions, and methods on objects, promise to satisfy certain post-conditions (or promises) if the inputs they are given satisfy the pre-conditions (or requirements) specified in the contract. Design by Contract also contains constructs for specifying loop-invariants and class-invariants, properties that should always hold during loops and for objects of a class, respectively. As-

---

[1]See http://eiffel.com/ and http://se.ethz.ch/ meyer/publications/online/eiffel/basic.html for a description of the language.

sertions (see below) are also usually available, to be interspersed with the program code.

Design by Contract is inspired by Hoare logic, and is essentially Hoare logic written in a certain style. However, most implementation of Design by Contract, such as that of Eiffel, and Code Contracts[2] in .Net, allow for arbitrary code in the specifications. The specifications are thus informal, unless a formalization is given for the programming language used.

Assertions

A common construct that is part of many popular programming languages, like C, Java and Python, is the *assert statement*. It is a way to state that some predicate should hold at a point in the program. Usually the predicate is an expression in the programming language, and is not supposed to alter the program state. See Figure 3.6 for an example.

```c
#include <assert.h>
#include <stdlib.h>

char *alloc_and_copy(char *src, int n) {
  assert(src != NULL);
  assert(n >= 0);
  char *dst = (char*) malloc(n);
  if (dst == NULL)
    exit(1);
  while (n--)
    dst[n] = src[n];
  return dst;
}
```

Figure 3.6. An example showing the use of `assert` in C. Note that the return value of `malloc` is checked to be non-null using an if-statement — this is a case that could happen. The assertions are used to check for cases that should never happen.

Assertions are distinct from the normal program flow, and not to be conflated with exceptions. Assertions check for properties that should always be true, anything else would be a programming error. Assertions can sometimes be turned of with a compiler or runtime flag.

Assertions are suitable for simpler specifications, and those more coupled to code. Assertions are the simplest of runtime verification specifications. If the languages they are part of are informal, the specifications they represent are informal as well.

---

[2]`http://msdn.microsoft.com/en-us/devlabs/dd491992.aspx`

3.2.2 Developing with Specifications

For verification in general, specifications can be written and used externally to the program. They can be used in specialized model-checking tools, in tools for theorem proving, etc.

Runtime verification requires that the specifications are accessible when building and running the program. At the very least, the program needs to be instrumented to expose the correct system model so that the specification can be verified. It is sometimes desired in runtime verification to do online verification, and then the specifications need to be available and embedded into the system. A few different approaches have been tried to support this.

Approaches to writing specifications can be divided into two parts: those that require you to manually mark code for verification, and those that inject the verification code from external specifications.

Rosenblum [16] uses specially annotated comments, written directly in the code. Bodden [17, 18] uses Java annotations, which are written at function and variable definitions, to mark code for verification. The programming language Eiffel has full language support for Design by Contract, with pre- and post-conditions, invariants, and more. These are written in direct proximity to the code to be verified. For simple cases it is common to write assertions in the program [19].

Other approaches, such as the ones taken by Jalili et al. in [20] and Barringer et al. in [13], use external specification files, and the program is instrumented at compile- or runtime for verification.

3.3 Verification against Specifications

Specifications for runtime verification are written so that programs can be verified against them — to see whether they follow the specification, or violate parts of it.

There are several ways to verify a program against its specification. A common one, used in [12, 17, 20, 13] among others, is to generate state machines from the specification. These state machines, sometimes called *runtime monitors*, operate with the input language of events emitted by the program. As the program executes, or a program execution is replayed offline, transitions are taken in the state machines, switching states. Violations against the specification can be described by failing to find a valid transition, or ending up in a fail state; Fulfilling a specification can be described by arriving at an accepting state.

Simpler verification approaches just instrument the code by adding assertions that check the properties of the specifications. This might be difficult to do with more complex specifications.

3.4 Code Instrumentation

For verification to work, the verifier needs access to events happening in the program. Such events can be functions called, statements executed, variables assigned, etc., depending on the system model of the specification language. The program needs to be instrumented for it to emit such events. This often means wrapping function calls and variable assignments in a "recording layer", which performs the desired action after logging the event. The events can then be passed on to the verification tools.

What follows are four major approaches used for program instrumentation.

3.4.1 Pre-processing the Code

Rosenblum [16] uses a pre-processor step in the C compilation setup to instrument code, where the specifications (called assertions by Rosenblum) are transformed from comments into regular C code. See Figure 3.7 for an example. The verification code is then compiled together with the program.

```
void swap(int *x, int *y)
  /*@
      assume x && y && x != y;
      promise *x == in *y;
      promise *y == in *x;
  @*/
{
  *x = *x ^ *y;
  *y = *x ^ *y;
  /*@
      assume *y == in *x;
  @*/
  *x = *x ^ *y;
}
```

Figure 3.7. An example showing instrumentation via special comments and code pre-processing. Example taken from [16].

3.4.2 Post-processing the Code

It is also possible to rewrite the compiled program, instrumenting the code after compilation. This way, the program needs no knowledge of the verification framework. Depending on the compiled objects, this can be more or less difficult. Binary executables and intermediate formats, such as Java bytecode or Common Inter-

mediate Language for the Common Language Infrastructure used by .Net, require somewhat different approaches.

3.4.3 Dynamic Code Rewriting

In many dynamic languages, such as Python, Ruby or JavaScript, it is possible to rewrite the code during runtime, which is sometimes called *monkey patching*. A function to be monitored could be rewritten, adding a lightweight wrapper that records all calls to it, and then passes on the call to the actual function.

3.4.4 Aspects

An interesting approach to code instrumentation is to use aspect-oriented programming. In aspect-oriented theory, a program should be divided into modules, each only dealing with their own *concern*. Logging, however, is a *crosscutting concern*, as it is used by several unrelated modules. The goal is to not scatter logging code across the modules, and to not tangle it with the modules' own logic. This can be done by defining the logging code as *aspects*, which consists of the logging code, called the *advice*, and a *point cut*, which is a formula describing when the advice should be executed. The possible execution points for a point cut are called *join points*. AspectJ[3] is the canonical framework for aspect-oriented programming.

Runtime verification is a typical case of a cross-cutting concern. Bodden [17] uses AspectJ in his runtime verification implementation.

Aspects in AspectJ are implemented as a post-processing step in the compilation process, adding wrapper code for handling the aspects.

---

[3]`http://www.eclipse.org/aspectj/`

Chapter 4

Introduction to Unit Testing

We discussed testing and in general in Section 2.3. Here we'll delve into more detail about unit testing in particular, and discuss how it works, and what the syntax is like.

In unit testing, a program is divided into small, separate parts — silos, with minimum dependencies to the rest of the system — and each unit is tested in isolation. Dividing a program into these kinds of units can be difficult, and proponents of unit testing argue that it requires a decoupled and modular system design.

Unit testing is quite young, perhaps having begun in earnest in the 90s, and it was popularized by the extreme programming (XP) movement[1]. Testing in general is very old.

Kent Beck introduced the style of the modern unit testing framework in his work on a testing framework for Smalltalk [21]. Together with Eric Gamma he later ported it to Java, resulting in *JUnit*[2]. Today, this has lead to *frameworks* in several programming languages, and they are collectively called xUnit [22]. Examples include, beside JUnit for Java: *unittest*[3] for Python, *NUnit*[4] and *xUnit.net*[5] for C# and .Net, *RSpec*[6] and the *Test::Unit*[7] package for Ruby, and *Jasmine*[8] for JavaScript.

When discussing testing, and unit testing in particular, we must mention the concept of test-driven development (TDD). Also made popular by XP, it consists of the cycle: 1) write a failing test; 2) make it pass by writing the simplest code you can; and 3) refactor — rewrite the code, cleaning it up and giving it a better structure. Tests here play the part of specifications for the units of the program.

---

[1]`http://www.extremeprogramming.org/`

[2]`http://www.junit.org/`

[3]`http://docs.python.org/library/unittest.html`

[4]`http://www.nunit.org/`

[5]`http://xunit.codeplex.com/`

[6]`http://rspec.info/`

[7]`http://www.ruby-doc.org/stdlib-1.9.3/libdoc/test/unit/rdoc/Test/Unit.html`

[8]`http://pivotal.github.com/jasmine/`

4.1 xUnit

The xUnit style of unit testing [22] has given rise to unit testing frameworks for many programming languages. Their structure are all based on the same concept, and since JUnit is the canonical implementation, and one of the first, implementation, we will use it for a short demonstration. See Figure 4.1. Jasmine, being younger and more "hip", has a slightly different syntax, but essentially the same structure. See Figure 4.2.

In JUnit, and xUnit, you run a *test suite* of *test cases*, which contain tests. The example in Figure 4.1, the test suite is implicitly created by JUnit, although it is possible to create it and control it your self. A *test runner* runs the test suite, reporting progress to the user. When the tests are finished, any errors are displayed.

In the example in Figure 4.1 has two tests, and methods to set up and tear down the tests *fixture*. These functions are usually called *setUp* and *tearDown*, respectively, and are called before and after each test. The fixture is the surrounding set of objects (environment) that the object under test requires to work properly.

Test written in the style of Figure 4.1 are traditional unit tests.

4.2 Mocking and Faking

A common issue when writing unit tests is that, to instantiate some object X, or to call some function Y, the program needs access to some other objects Z. Z might be something simple, which we can easily create in the test. It might also be a network or database connection, or something doing heavy calculation, or just something complex.

One way to work around this is to create fake objects (also called mock-, fake- or spy objects). A fake network connection has the same interface as a real network connection, but calling it does not actually transmit anything anywhere, and it might return pre-defined, hard coded data. Fake objects could save what actions are taken upon them, and the test could then verify that these are according to expectations.

4.3 Expectations

Instead of writing fake objects, we can create a mock object and pre-record what actions we expect to be taken upon them. This is called writing *expectations* [23]. A simple example of expectations is shown in Figure 4.3.

Figure 4.3 shows a test of a fictional shop. The test tests only one thing, the fill method of the Order object, but it requires a Warehouse object, for access to the inventory. We supply a mock Warehouse, with expectations on which methods should be called on it, with which arguments and what they should return.

An expectation follows a simple pattern:

- A function, with an optional object, which is expected to be called.

```java
// required imports removed for brevity

public class TestSomeClass
    extends TestCase {
  private Environment;

  @Before
  public void setUp() {
    // setup the fixture for each test
    Environment = new Environment();
  }

  @After
  public void tearDown() {
    // clean up the fixture, free memory, etc.
  }

  @Test
  public void testSimpleAddition() {
    // use the language assertion construct
    assert 1+1 == 2
    // use JUnit's assertion functions
    assertEquals(4+7, 11)
  }

  @Test
  public void testThatDoWorkReturnsX() {
    // do setup for this test
    Target t = new Target(...);
    // exercise the object under test
    t.doWork(...);
    // do verification
    assert t.getValues() == x;
  }
}
```

Figure 4.1. An example of unit testing syntax, written Java as a test case for JUnit. All tests are marked with the **@Test** annotation. The **assert** keyword and functions like **assertEquals** are used to verify the correctness of the outputs.

```
describe("simple test of the adder module", function() {
  var adder;

  beforeEach(function() {
    adder = new Adder();
  });

  it("should return the sum of two values", function() {
    expect(adder.add(2, 7)).toEqual(9);
  });
});
```

Figure 4.2. An example of unit testing syntax, written in JavaScript and Jasmine. It consists of a `describe` test suite, and one test case `it`, called a specification in jasmine parlance. The `expect(...).toEqual(...)` style is used for verification.

- An invocation count of how often the function is expected to be called.

- Expected arguments for the function call. These can be explicit values, or generic types, or rules defining the acceptable values.

- The return value and modifications to the global state; what should happen when the function is called.

- When the function call should happen, e.g. in what sequence of function calls, in what global state.

There are two common ways of specifying expectations: recording and explicit specification. Figure 4.3 shows an example of how to explicitly specify expectations.

When recording expectations, you create a mock object and call the expected functions, with expected arguments and return values, in the expected order. Then you set the mock into replay mode, and it will replay the recorded expectations, and verify that they occur correctly.

There are several frameworks for working with expectations, such as jMock[9] for Java, Rhino Mocks[10] for .Net and Ludibrio[11] for Python.

With both runtime verification and unit testing introduced, a combination of the two will be the main subject in the next two chapters.

---

[9]http://www.jmock.org/
[10]http://ayende.com/wiki/Rhino+Mocks.ashx
[11]https://github.com/nsigustavo/ludibrio/

```java
public class OrderInteractionTester
    extends MockObjectTestCase {
  private static String TALISKER = "Talisker";

  public void testFillingRemovesInventoryIfInStock() {
    // setup - data
    Order order = new Order(TALISKER, 50);
    Mock warehouseMock = new Mock(Warehouse.class);

    // setup - expectations
    warehouseMock.expects(once())
      .method("hasInventory")
      .with(eq(TALISKER),eq(50))
      .will(returnValue(true));
    warehouseMock.expects(once())
      .method("remove")
      .with(eq(TALISKER), eq(50))
      .after("hasInventory");

    // exercise
    order.fill((Warehouse) warehouseMock.proxy());

    // verify
    warehouseMock.verify();
    assertTrue(order.isFilled());
  }
}
```

Figure 4.3. An example of expectations, written using jMock and JUnit. Example taken from [23].

Chapter 5

Approach

With all this background overview and introductions out of the way it is time for the core and main substance of this report.

5.1 Introduction

As stated in Section 1.1, the objective of this report is to investigate whether it is possible to do runtime verification with specifications written in the target program's programming language, structured similar to unit tests. To find a solution for this, there are four issues we need to address:

1. How should the syntax for the specifications be defined, so that it looks similar to that of unit tests, but works for runtime verification? Which language should be used? Which unit testing framework to take inspiration from?

2. How should the program be instrumented to monitor the system, to expose the appropriate events and data, and to build the system model?

3. How will this be used to verify the system against the specification? Should the verification be done online or offline? Which techniques should be used to verify the monitored system against the specification?

4. How can the resulting approach be provided with a formal foundation?

This report is a documentation on how to solve these issues. The following sections are each dedicated to one issue, and shows a proof-of-concept of these ideas. The implementation, called *pythonrv*, can be found online[1].

5.1.1 Definitions

The following definitions will be used through the next few sections.

---

[1] `https://github.com/tgwizard/pythonrv`

- A *specification* is a construct that determines the correct behaviour of a program. It could be a document, describing the programs functionality, or a set of inputs and outputs, describing the correct results of the program's computation on that set. It could be a reference implementation[2]. A *formal specification* is a mathematical construct that can be used in verification proofs to show that a program works correctly, i.e. according to its specification.

- In *pythonrv* a *specification function* is a Python function describing a specification, which *pythonrv* can use for verification of the program.

- A specification function *monitors* points of the program, and the points being monitored are called *monitorees*. Currently only functions are supported as monitorable points.

5.1.2 Implementation Language Discussion

During the development of this proof-of-concept, the biggest factor in deciding what language to use was how it would assist in instrumentation. Instrumentation is discussed in Section 5.3. The language should also be in wide use, support quick development, and have an active testing culture.

Easy access to a non-trivial and actively used system for real-world testing would be a plus. More on this in Chapter 6.

Python[3], among several languages, fits these criteria, and was chosen as the implementation language. Python is a dynamic programming language, commonly running inside a virtual machine. Perhaps the most striking feature of Python is its use of indentation as block delimiters. An increase in indentation starts a new block; a decrease marks the end of the current block.

The name *pythonrv* clearly reflects that it is a framework for runtime verification in Python.

The syntax, described in the next section, of the *pythonrv* specifications are naturally heavily influenced by the fact that they are written in Python.

5.2 Syntax

The canonical framework for doing unit testing in Python is the *unittest* framework that is included in all modern versions of Python. Not much development has happened on it in the last years. Many new frameworks have have spawned, such as PyUnit, Nose and py.test. They build upon the style of unittest and mostly add new miscellaneous features, such as better test reporting. The original structure of the unit tests is still prevalent — unittest builds on the xUnit style of unit testing, discussed in Chapter 4.

The next section will illustrate the syntax of *pythonrv*.

---

[2]For instance, the only specification for Python is the canonical CPython implementation. Python is defined as "what CPython does".

[3]`http://www.python.org`

5.2.1   General Structure of a Specification

The structure of a informal *pythonrv* specification function is shown in pseudo-code
in Figure 5.1. For the structure of formal specifications, see Section 5.5. The next
section illustrates the *pythonrv* syntax for informal specifications with three concrete
examples

```
import pythonrv module
import modules with functions to be monitored

annotate the spec with its monitorees
define specification function spec(event):
  taking a special event argument

  with a body of arbitrary Python code
  that verifies that the provided event is correct
  and throws an AssertionError exception when it is not
```

Figure 5.1.   The structure of a *pythonrv* specification. Note that the Python
`assert` statement throws an `AssertionError` exception when its argument evaluates
to `False`.

5.2.2   Three Examples of Informal Specification Functions

```
1    from pythonrv import rv
2    import fibmodule
3
4    @rv.monitor(func=fibmodule.fib)
5    def spec(event):
6      assert event.fn.func.inputs[0] > 0
```

Figure 5.2. A specification that monitors the function `fib` in the module `fibmodule`.
The monitored function is, locally to the specification function, aliased as `func`. The
specification asserts that the first input to the monitored function is always greater
than zero.

The example in Figure 5.2 shows the basics of a *pythonrv* specification, writ-
ten as a specification function. Line 1 imports the `rv` module from the `pythonrv`
package. On line 2 it imports the module containing the function to be monitored.
Line 5 defines the specification as an ordinary Python function called `spec`, taking
one argument, `event`. The instrumentation is done line 4 by using the *function*

decorator[4] `rv.monitor`. `rv.monitor` declares that the function `fib` in `fibmodule` should be monitored, and, whenever `fib` is called, `spec` should be called as well.

The specification function itself consists of any valid Python code. It is passed a special argument, `event`, which gives the specification function access to data about the current event. On line 6, the array of input arguments used to call `fib` is accessed to check that the first argument is greater than zero.

The specification function in Figure 5.2 will be called upon every invocation to `fibmodule.fib`.

```
1    from pythonrv import rv
2    import mymodule
3
4    @rv.monitor(foo=mymodule.foo,
5      bar=mymodule.bar)
6    def spec(event):
7      if event.fn.foo.called:
8        # the foo function was called
9        # either the size of the event history
10       # is 1 - this is the first event - or
11       # the previous event was a call to bar
12       assert len(event.history) == 1 \
13         or event.prev.fn.bar.called
14     elif event.fn.bar.called:
15       # the bar function was called
16       # assert that previous event
17       # was a call to foo
18       assert event.prev.fn.foo.called
```

Figure 5.3. A specification that monitors two functions, `mymodule.foo` and `mymodule.bar`. It asserts that calls to the two functions alternate; that no two calls to `foo` occurs without a call to `bar` in between, and vice versa. The first call has to be to `foo`.

Figure 5.3 shows how a specification function can monitor two functions. The specification function will be called whenever either of the monitored functions are called. Which function was called can be determined from the `event` argument, as is done on lines 7 and 14. It is the `called` attribute of a function in the `event.fn` structure that allows for this.

The example also shows how the specification can access a history of previous events - events that it has handled in the past. `event.history` is a list of all events that has occurred that this specification monitors. The last element is the

---

[4]See Section 5.3 for an explanation of function decorators and `rv.monitor`.

current event, and the next-to-last element is the previous element, which can also
be accessed as `event.prev`.

```
1    from pythonrv import rv
2    import mymodule
3
4    @rv.monitor(foo=mymodule.foo,
5      bar=mymodule.bar, baz=mymodule.baz)
6    def spec(event):
7      if event.fn.foo.called:
8        # add function to be called on
9        # next event
10       event.next(followup)
11       event.finish()
12     else:
13       # verification has failed
14       # similar to assert False
15       event.failure()
16
17   def followup(event):
18     if event.fn.bar.called:
19       event.success()
20     elif event.fn.baz.called:
21       assert event.fn.baz.inputs[0] == True
22       # call this function on next event
23       # as well
24       event.next(followup)
25     else:
26       event.failure()
```

Figure 5.4. A more complex example: A specification function that monitors three
functions, `foo`, `bar` and `baz`, and makes sure that `foo` is called first, then any number
calls to `bar` with the first argument as `True`, and then finally a call to `bar`. After that,
any calls are allowed — the specification function will not be used in verification any
longer.

Figure 5.4 shows a more advanced example, in which the `next` function of the
`event` argument is used. `event.next` allows the specification function to add more
specification functions (possibly implemented as closures or lambdas) to be executed
when the next event occurs.

On line 9 the function `followup` is added to be executed on the next event.
Since `followup` is added in this way — as a "oneshot" specification function — it
needs to add itself using `next` for verification on subsequent events. This is done

on line 22.

Figure 5.4 also shows how a specification function can turn its verification off — i.e. unsubscribe from future events. `event.finish` and `event.success` are essentially the same, and unsubscribes without further errors. `event.failure` can be thought of as a combination of `event.finish` and `assert False` (which always fails).

5.2.3 Capabilities and Limitations

The examples above show the main capabilities of *pythonrv* specifications. A few minor details were left out, such as how to specify how much history should be saved for a specification, or how to label specifications with error levels, so that different actions can be taken depending on which specification function fails. This is described on the website for *pythonrv.*

5.3 Instrumentation

The previous section showed how *pythonrv* specification functions can be written. This section will describe how these functions can jack themselves into the ordinary control flow of the program and gain access to the function call events and their arguments and associated state.

Instrumentation is done through the `rv.monitor` *function decorator* in *pythonrv.* A Python function decorator is similar to attributes in .Net and annotations in Java. It is essentially a function that takes in a function and returns a function, possibly modifies it, or uses it in some way (decorates it) in the process. This is used throughout Python to, for instance, turn functions into static or class methods. Figure 5.5 shows an example function decorator definition, and Figure 5.6 shows how to use it.

`rv.monitor` first takes arguments specifying what functions should be monitored, and then the specification function itself.

In Python, almost all[5] functions belong to a container of some sort — a class, an object, or a module. In Figure 5.6 the functions `func_a` and `func_b` belong to the module `test` (the module's name is the same as that of the file containing the code). These containers are essentially dictionaries (*dicts* in Python parlance) of key-value pairs, where the keys in this case are function names and the values are objects representing the function code. (There are other types of values in these containers as well, which we can ignore).

The instrumentation in *pythonrv* works as follows. First, a wrapper function is defined for each function to be monitored (for each monitoree). This wrapper function's main purpose is to call the specifications attached to the monitored function,

---

[5]This is not true of closures — functions defined inside other functions. These functions cannot be directly referenced or modified from outside the defining function. *pythonrv* does not (as of writing) support monitoring of closures.

```
1   # function_decorator.py
2   # the function decorator
3   def decorator(decoratee):
4     # define the closure ("inner function")
5     def wrapper():
6       print "before",
7       # call the decorated function
8       ret = decoratee()
9       print "after",
10      return ret
11    # return the closure
12    return wrapper
```

Figure 5.5. An example of how to define a function decorator.

```
1   # test.py
2   from function_decorator import decorator
3
4   def func_a():
5     print "a",
6
7   func_a()
8   # output: a
9
10  # decorate func_a
11  func_a = decorator(func_a)
12  func_a()
13  # output: before a after
14
15  # decorate func_b - equivalent
16  # to the decoration of func_a
17  @decorator
18  def func_b():
19    print "b",
20
21  func_b()
22  # output: before b after
```

Figure 5.6. An example of how to use the function decorator from Figure 5.5.

and then call the monitored function itself. The wrapper also does some argument copying and such, to prevent side-effects in the specifications from interfering with the monitored function. The container of the monitored function is then extracted, and the reference to the monitored function is overwritten with a reference to the wrapper. See Figure 5.7 for an overview.

The implementation of the instrumentation code in *pythonrv* is more optimized than this - for instance, only ever one wrapper per monitoree is created, independent of the number of specifications that want to monitor it.

```
1    # rv.py
2    def monitor(monitorees, specification):
3      for monitoree in monitorees:
4        # define a wrapper for each monitoree
5        def wrapper(*args, **kwargs)
6          event = create_event(...)
7
8          # call specification
9          specification(event)
10
11         # call the actual function - the
12         # monitoree
13         return monitoree(*args, **kwargs)
14
15       # overwrite the monitoree in its container
16       container = get_container(monitoree)
17       setattr(container, monitoree.name, wrapper)
```

Figure 5.7. An overview of the *pythonrv* instrumentation process, written in pseudo-Python. This is just for illustrative purposes and not how *pythonrv* actually does the instrumentation.

5.4 Verification

In *pythonrv*, verification is quite simple. The specification functions are executable, and executing them on the appropriate events, providing access to the current data, verifies that the specification they represent is followed.

Specification functions notify verification violations, that the specifications are not followed, by raising exceptions of the type `AssertionError`. These exceptions are raised when the `assert` statement fails. They can also be raised manually: `raise AssertionError('error message')`.

The verification is performed online, during the program execution. Specifications are verified for all calls to function they monitor unless they explicitly remove

themselves by calling one of `event.finish`, `event.success` and `event.failure` (described in Section 5.2).

5.4.1 Dealing with Errors

Whenever a specification violation occurs, and an `AssertionError` is raised, it is passed to an *error handler*. There are two built-in error handlers: One, the default, that re-raises the exception, and thus crashes the program[6], and a second which just logs the error message, using the standard Python logging module.

It is possible to write custom error handlers for *pythonrv*. See the website for how.

5.4.2 Offline Verification

The current verification approach in *pythonrv* is to perform it online. This obviously affects the performance of the program under test. Offline verification could be used to mitigate this, removing all overhead but for the required recording layer.

To do offline verification in *pythonrv* the events and their associated data would need to be saved (serialized) and replayed outside the context of the running program.

5.5 Formal Foundation

The purpose of a formal foundation for a verification approach is so that the specification writer can reason mathematically about her specifications, showing their semantical correctness using proof techniques. To do this, the specifications used for verification need some sort of mathematical representation. Different kinds of automata are often used. A model of the system, a system model, is also required. In this case the system model is the sequence of that occur during the program's execution, with their associated data (arguments, object state, global state, etc.). This is described in more detail in Section 5.5.2

A seemingly insurmountable problem quickly arises when attempting to give such a mathematical representation to the *pythonrv* specifications described in Section 5.2. *pythonrv* specifications are written as ordinary Python functions and, as such, are difficult to formalize. The Python programming language is rather informal - one implementation of it, CPython, serves as the reference implementation. There are no other specifications or formal semantics for Python[7].

One way to go around this is to define formal semantics for a subset of Python, which is done in the following sections. This leads to a way to reason mathematically with and about specifications written in this subset — see Section 5.5.3.

---

[6]Unless some other part of the program, higher up the call stack, suppresses the exception.

[7]The development of Python is organized mainly through the Python Enhancement Proposal (PEP) process. PEPs are design documents for new features, informally describing their rationales and how they work.

5.5.1 Terminology and Definitions

We begin with some definitions and terms to make the following sections simpler.

- A *formal specification function* is one of three basic functions described in Section 5.5.3, or a composition of them as described in Section 5.5.4.

- Formal specification functions have *composition points* where they can be combined with other formal specification functions. A formal specification function can have zero or more composition points. A composition point can be *open* or *closed* — open composition points can be used in composition, while closed have already been used. Composition points can be *required* or *optional.*

- A formal specification function can be either *complete* or *incomplete*. A complete formal specification function is a valid specification. An incomplete formal specification function is not a valid specification, but can, with composition, become complete. Complete formal specification functions have no open required composition points; incomplete formal specification functions have at least one.

- Composition is described by the $\circ$ operator. Let f, g and h be formal specification functions. Let f have one composition point, g two, labeled a and b, respectively, and h none. A valid composition would be: $s = f \circ ((g \circ_a h) \circ_b h)$. s would be a complete formal specification function, as it is composed of formal specification functions, and it has no open required composition points (or optional ones).

- Every formal specification function s can be represented by a nondeterministic finite automata $A(s) = (Q, P, q_0, S, F)$. Each such automata consists of a set of states Q and transitions $(a, b, E) \in P$ between them, where a is the start state, b is the end state, and E is a *label*. An automata has one initial state q_0, a set of *success states S* and a set of *fail states F*.

- A state cannot be in both S and F: $S \cap F = \emptyset$. Success states are depicted as *accepting states* in the automata. Fail states are usually called *fail*, f, etc.

- Each transition in the automata has a label — an expression — which is described in detail in Section 5.5.2. The label is taken from the expressions in the Python functions, as shown in Section 5.5.3. The label $\top$ denotes "any and all expressions".

- The set of *initial transitions T*, $(q_0, q', E) \in T$, $T \subseteq P$ are especially important for composition, see Section 5.5.4. All initial transitions start from q_0, go to some state q', and have some label E.

Notation: Let a_E be an *assert* specification asserting the expression E, let n be a *next* specifications and let i_E be an *if* specification with the expression E guarding the *then* composition point. Let s be any specification. Appending subscripts and $'$ denotes different specifications or expressions of the same type. The E subscript can be omitted if irrelevant to the task at hand.

5.5.2 Semantics

The semantics for the specifications as expressed in their Python format, even composed only from the basic specification functions in Section 5.5.3, is quite straightforward. The trick is to preserve this semantics when transforming the specifications into automata, while at the same time allowing for mathematical reasoning to deduce the correctness of the specifications. This requires two parts. First, a description of the system model. Second, a set of rules to deduce whether a specification would accept or reject an instance of such a model.

The system model

The system model is quite simple. It consists of a sequence of *events* of function calls, $S = (\alpha_0, \alpha_1, \ldots, \alpha_n)$. Each event $\alpha_i = (\lambda_i, \delta_i)$ is a two-part structure: The function called, λ, and the associated arguments and state, δ. Since the system model is an abstraction of an execution of the system (the target program), it is necessarily finite, but of arbitrary length, and possibly extendible through continued execution of the program.

The system model can be viewed as a directed acyclic graph (actually, a tree), where all nodes but the last have an edge. The first edge is labeled α_0, the second is labeled α_1, etc. See Figure 5.8 (a) for an example.

The product of a system model and a specification

In the same way that the system model can be viewed as a graph, so can a specification automata, see Figure 5.8 (b). Specification automata are defined and discussed more in sections 5.5.3 and 5.5.4. The product $Z = M \times C$ of a system model M and a specification automata C is the main construct used to reason about the correctness of the specification. An example can be seen in Figure 5.8 (c).

The product Z will be a tree of depth $min(depth(M), depth(C))$ (where the depth of a specification automata containing cycles is considered infinite).

First, the two initial states of M and C are merged, resulting in a new state (q_0, s_0). All transitions T from q_0 are merged with the (single) transition (s_0, s_1, α_0) from s_0. This results in a new set T_{q_0, s_0} of transitions from (q_0, s_0):

$$T_{q_0, s_0} \quad = \quad \{((q_0, s_0), (q, s_1), \alpha_0 \models E) \,|\, (q_0, q, E) \in T\}$$

The result is a set of second-level states, $Q_2 = \{(q, s_1) | (x, (q, s_1), E) \in T_{q_0, s_0}\}$, and the process continues. The transitions from all specification states q in Q_2 are merged with the transition (s_1, s_2, α_1) from s_1, generating a set of third-level states, Q_3. This terminates when the depth of either the specification or the system model is reached.

A state (q, s) of the product is a success state if q is a success state, and a fail state if q is a fail state.

See Figure 5.8 for an example.

Evaluating specification-model product label expressions

The transition labels for the product of a specification automata and a system model is of the form (the second line is a expanded form of the first):

$$\begin{aligned} \alpha &\models & \text{expression } E \text{ over } \alpha \\ (\lambda, \delta) &\models & \text{expression } E \text{ over } \lambda \text{ and } \delta \end{aligned}$$

The expression E is translated directly from Python expressions in the specification function. An incomplete sketch of the translation procedure, shown in Figure 5.9, is just syntax replacement. The resulting specification expression has the same semantics as the Python expression. In the same way that the Python specification operates on an event, a specification expression is evaluated to true ($\top$) or false ($\bot$) using the values of an event α.
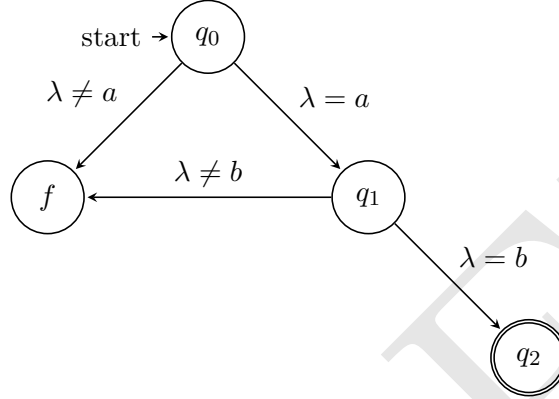
Verification

A product Z of a system model and a specification can be used to determine whether that specification would accept or reject the system model — or not know to do either or, yet.

1. Compute the product Z of the specification C and the system model M, as described above.

2. Traverse Z, starting at the root node, following all transitions which labels evaluate to true.

   a) If a fail state is reached, the system models violates the specification.

   b) If only success states are reached, the system model satisfies the specification.

   c) If some traversal ends in a state that is neither a fail or a success state, the system model could, if it were extended, either satisfy or violate the specification.
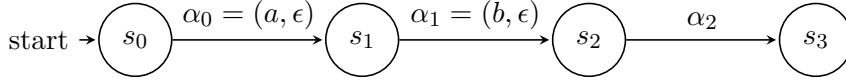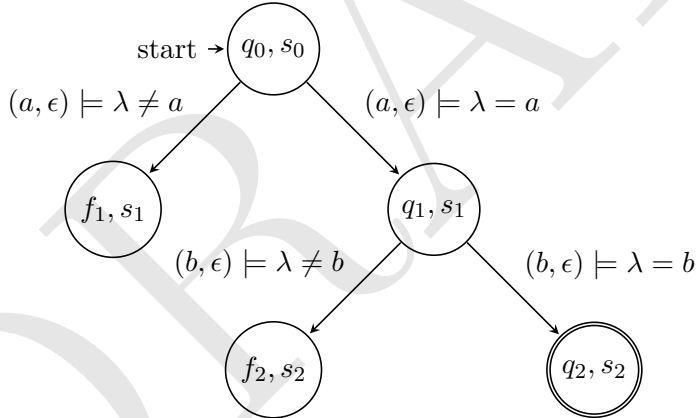
(a) The system model M consisting of two function calls, to a and b, and an unspecified event α_2.

(b) The specification automata C. This automata is designed to accept system models that consists of one call to a followed by a call to b. After that, verification is complete and any events are accepted. f is a fail state and q_2 is a success state.

(c) The product of the model and the specification, $M \times C$. (f_1, s_1) and (f_2, s_2) are fail states; (q_2, s_2) is a success state.

Figure 5.8. Deduction example.

Formal *pythonrv* specifications are verified using this procedure, although in an incremental manner. The verification is done online, so as soon as a new event is added to the system model, it is combined with the specification automata.

We now go on to describe the actual syntax for these formal specification functions.

$$
\begin{aligned}
PE(\texttt{True})) &= \top \\
PE(\texttt{False})) &= \bot \\
PE(\texttt{event.fn.x.called}) &= \lambda = x \\
PE(\texttt{event.called\_function == x}) &= \lambda = x \\
PE(\texttt{event.fn.x.property}) &= \delta(x.\texttt{property}) \\
PE(\texttt{not a})) &= \neg PE(\texttt{a}) \\
PE(\texttt{a == b})) &= PE(\texttt{a}) = PE(\texttt{b}) \\
PE(\texttt{a != b})) &= PE(\texttt{a}) \neq PE(\texttt{b}) \\
PE(\texttt{a and b})) &= PE(\texttt{a}) \wedge PE(\texttt{b}) \\
PE(\texttt{a or b})) &= PE(\texttt{a}) \vee PE(\texttt{b}) \\
PE(A\ Python\ function\ or\ property) &= The\ return\ value
\end{aligned}
$$

Figure 5.9. A sketch of the translation procedure of Python expressions to specification expressions. *PE* is the mapping function, replacing the Python syntax with mathematics syntax.

5.5.3 Python Subset for Formal Specification Functions

The subset of Python that can be used in formal *pythonrv* specification functions consists of three composable specification functions: *assert*, *next* and *if*; See Figure 5.10. A complete and verifiable formal specification function also needs some boilerplate, shown in Figure 5.11.

```
1   def spec():
2     a = make_assert(E)
3     tail = # optional composition point 'tail'
4     return a + tail
5
6   def next():
7     x = # required composition point 'next'
8     n = make_next(x)
9     tail = # optional composition point 'tail'
10    return n + tail
11
12  def if():
13    then = # required composition point 'then'
14    els = # optional composition point 'else'
15    i = make_if(E, then, else)
16    tail # optional composition point 'tail'
17    return i + tail
```

Figure 5.10. The three basic formal specification functions.

```
 1    from pythonrv import rv
 2    from pythonrv.formalrv import (formal_spec,
 3      make_assert, make_next, make_if)
 4
 5    monitorees = # list of monitorees
 6    @rv.monitor(monitorees)
 7    @formal_spec
 8    def spec():
 9      # spec is a formal specification function
10      # - assert, next or if
```

Figure 5.11. The boilerplate code for formal *pythonrv* specification functions that allow verification and instrumentation to take place.

The label E used in the assertion and if statement (lines 2 and 11) denotes any idempotent, immutable, valid Python boolean expression. A technical detail is that if the expression needs to access any properties of the `event` argument, they need to be of the form `lambda event: E`. The previous section described how these expressions are used in formal verification.

The three basic formal specification functions correspond to simple, nondeterministic, finite automata, which are depicted in Figure 5.12. The basic formal specifications can be combined by composition into larger, more complex specifications. Details on the basic formal specification functions and their composition is described in the next section. Note that the *assert* formal specification function, and all complete compositions of the three basic formal specification functions, are valid *pythonrv* specification functions.

5.5.4 Transformation to Automata and Rules for Composition

The idea is to use composition to build more complex and interesting specifications from the three basic formal specification functions. Composition is defined inductively, preserving the semantics from Section 5.5.2.

There are no implicit precedence rules for the ∘ composition operator; parentheses are required.

Composition of the *assert* specification, and composition with the *tail* composition point

The *assert* specifications are the only basic formal specification functions that are complete, without the need to compose them with other specifications. The automata $A(a_E)$ for a standalone *assert* specification a_E, depicted in Figure 5.12 (a), is defined as:

37
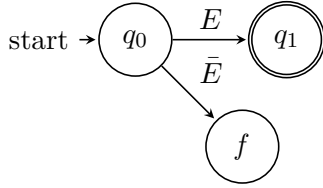
(a) A generalized *assert* specification, assert-
ing the expression E. q_1 is a success state; f
is a fail state.
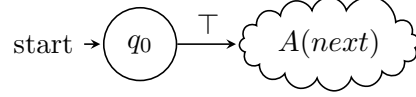
(b) A template for a *next* specification.
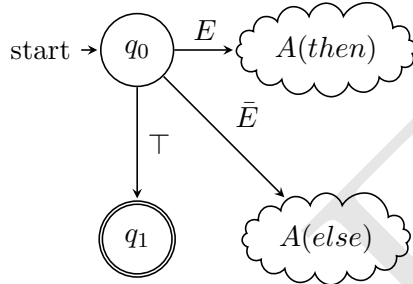
(c) A template for an *if* specification.

Figure 5.12. Sketches of automata for the three basic formal specification functions.
The squiggles around $A(next)$, $A(then)$ and $A(else)$ denote that some composition
is required there.

$$A(a_E) \quad = \quad (\{q_0, q_1, f\}, \{(q_0, q_1, E), (q_0, f, \bar{E})\}, q_0, \{q_1\}, \{f\})$$

If $\bar{E}$ is true, the fail state f will be reached, and the specification has been
violated.

assert specifications also have, together with all basic formal specification func-
tions and compositions thereof, at least one open composition point, the *tail* com-
position point. Compositions using the *tail* composition point are commutative:
$s \circ_{tail} s' = s' \circ_{tail} s$. Composition using the *tail* composition point is essentially
just a merge of the initial states of the two specifications, making the initial transi-
tions of both specifications go out from the same state q_0, and merging the success
states and fail states of the automata. Given two specifications s and s', where:

$$\begin{aligned} A(s) \quad &= \quad (Q_s, T_s \cup R_s, q_{0s}, S_s, F_s) \\ A(s') \quad &= \quad (Q_{s'}, T_{s'} \cup R_{s'}, q_{0s'}, S_{s'}, F_{s'}) \end{aligned}$$

Then:

$$
\begin{aligned}
A(s \circ_{tail} s') &= (Q, P, q_0, S_s \cup S_{s'}, F_s \cup F_{s'}) \\
Q &= \{q_0\} \cup Q_s \cup Q_{s'} - \{q_{0s}, q_{0s'}\} \\
P &= T \cup R_s \cup R_{s'} \\
T &= \{(q_0, q', E) \mid (q, q', E) \in (T_s \cup T_{s'})\}
\end{aligned}
$$

An example for composing two *assert* specifications is shown in Figure 5.13. The resulting automata is unnecessarily complex, and can be simplified to a smaller automata with the same semantics, as seen in Figure 5.14.
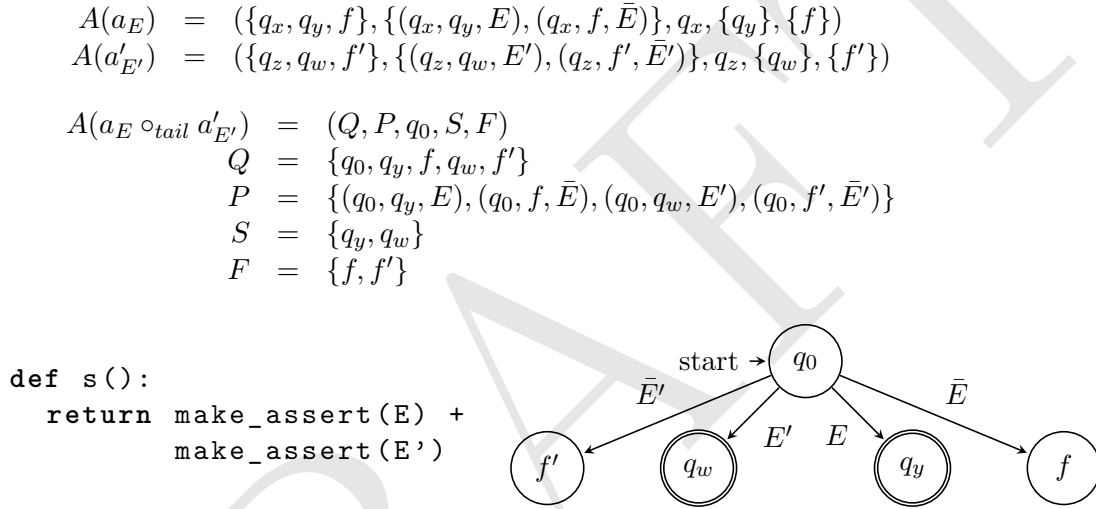
$$
\begin{aligned}
A(a_E) &= (\{q_x, q_y, f\}, \{(q_x, q_y, E), (q_x, f, \bar{E})\}, q_x, \{q_y\}, \{f\}) \\
A(a'_{E'}) &= (\{q_z, q_w, f'\}, \{(q_z, q_w, E'), (q_z, f', \bar{E}')\}, q_z, \{q_w\}, \{f'\})
\end{aligned}
$$

$$
\begin{aligned}
A(a_E \circ_{tail} a'_{E'}) &= (Q, P, q_0, S, F) \\
Q &= \{q_0, q_y, f, q_w, f'\} \\
P &= \{(q_0, q_y, E), (q_0, f, \bar{E}), (q_0, q_w, E'), (q_0, f', \bar{E}')\} \\
S &= \{q_y, q_w\} \\
F &= \{f, f'\}
\end{aligned}
$$

```
def s():
  return make_assert(E) +
         make_assert(E')
```


Figure 5.13. An example showing a composition of an *assert* specification a_E with another *assert* specification $a'_{E'}$. Only the resulting $A(a_E \circ_{tail} a'_{E'})$ automata is shown. q_y and q_w are success states; f and f' are fail states.

$$(\{q_0, q_1, f\}, \{(q_0, q_1, E \wedge E'), (q_0, f, \bar{E} \vee \bar{E}')\}, q_0, \{q_1\}, \{f\})$$
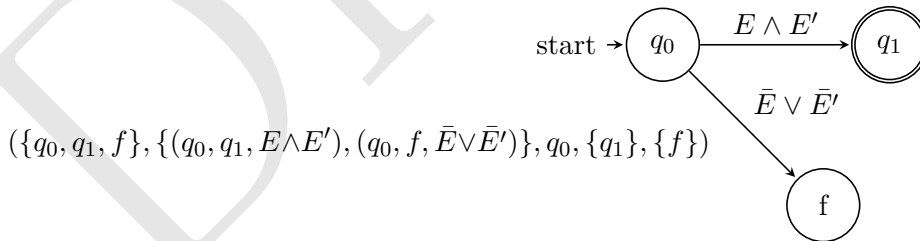
Figure 5.14. A simplified version of the automata from Figure 5.13, semantically identical. The success states have been merged into q_1, and the fail states into f.

Composition of the *next* specification

The *next* specification functions are the specifications that deal with time. *next* specifications have two composition points: one appropriately called *next*, which is

39

required, and one called *tail*, which is optional. Composition using the *tail* composition point was described above.

Composition with the *next* composition point, $n \circ_{next} s$ with $A(s) = (Q, P, q_0, S, F)$ is as follows:

$$\begin{aligned} A(n \circ_{next} s) &= (\{q_0'\} \cup Q, T \cup P, q_0', S, F) \\ T &= \{(q_0', q_0, \top)\} \end{aligned}$$

This is illustrated in Figure 5.15.

Also note that composition using the *next* composition point is associative, as shown in Figure 5.16.
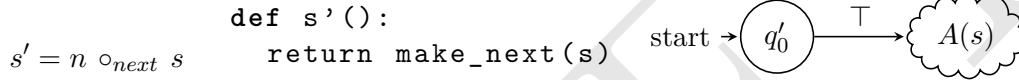
$s' = n \circ_{next} s$

```
def s'():
    return make_next(s)
```

start $\rightarrow$ q_0' $\xrightarrow{\top}$ $A(s)$

Figure 5.15. The composition of a *next* specification with any specification s, using the *next* composition point.

$$\begin{aligned} (n \circ_{next} s_1) \circ_{tail} (n' \circ_{next} s_2) &= n \circ_{next} t \\ t &= s_1 \circ_{tail} s_2 \end{aligned}$$
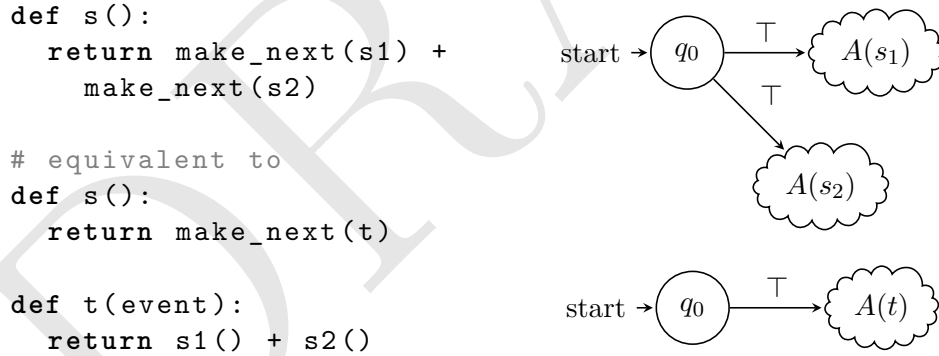
```
def s():
  return make_next(s1) +
    make_next(s2)

# equivalent to
def s():
  return make_next(t)

def t(event):
  return s1() + s2()
```

start $\rightarrow$ q_0 $\xrightarrow{\top}$ $A(s_1)$
$\xrightarrow{\top}$ $A(s_2)$

start $\rightarrow$ q_0 $\xrightarrow{\top}$ $A(t)$

Figure 5.16. The associativity of the *next* composition point in a *next* formal specification function.

Composition of the *if* specification

if specifications have three composition points: one required, *then*, and two optional, *else* and *tail*. Composition using the *tail* composition point was described above.

In an *if* specification i_E we consider the expression E as a guard for the *then* composition point, and $\bar{E}$ as a guard for the *else* composition point.

The composition essentially becomes to add the guard E to the labels of all initial transitions for the automata at the *then* composition point, and $\bar{E}$ to the labels of all initial transitions for the automata at the *else* composition point. The guards, E and $\bar{E}$ are added as parts of a conjunctive.

Let s_1 be the specification attached to the *then* composition point and an optional s_2 attached to the *else* composition point, and $A(s_1) = (Q_1, T_1 \cup R_1, q_{10}, S_1, F_1)$, $A(s_2) = (Q_2, T_2 \cup R_2, q_{20}, S_2, F_2)$. Then:

$$
\begin{aligned}
A((i_E \circ_{then} s_1) \circ_{else} s_2) &= (Q, P, q_0, S, F) \\
Q &= Q_1 \cup Q_2 \cup \{q_0, q_1\} - \{q_{10}, q_{20}\} \\
P &= T_1' \cup T_2' \cup R_1 \cup R_2 \cup X \\
T_1' &= \{(q_0, q, E \wedge E') \,|\, (q_{10}, q, E') \in T_1\} \\
T_2' &= \{(q_0, q, \bar{E} \wedge E') \,|\, (q_{20}, q, E') \in T_2\} \\
X &= \{(q_0, q_1, \top)\} \\
S &= S_1 \cup S_2 \cup \{q_1\} \\
F &= F_1 \cup F_2
\end{aligned}
$$

If the *else* composition point is left out, then $Q_2 = T_2 = R_2 = S_2 = F_2 = \emptyset$. X is the fall-through transition to the success state q_1, which can always be reached. This will allow for successful verification when E is false and there is no else clause.

With these constructions in mind, the next section shows a few functioning examples.

5.5.5 Examples of Formal Specifications

Here follows two examples of formal *pythonrv* specifications, both in their specification function form, written in Python and shown in Figure 5.17 and Figure 5.19, and as automata, shown in Figure 5.18 and Figure 5.20.

The first example, Figure 5.17 and 5.18, shows a simple specification that makes sure that the first argument to the function `fib` in the module `fibmodule` is always positive. The main difference with the similar informal specification in Figure 5.2 is that here we have to explicitly create a loop by composing with a *next* specification, pointing back to `spec`. This is required so that not only the first event is verified.

The example in Figure 5.19 is more complex, as can be seen by the sprawling automata in Figure 5.20. The specification verifies accepts only interleaving calls to two functions, starting with `mymodule.foo`, then to `mymodule.bar`. This is represented by the loop between the q_0 and r_0 states in the automata. Because the labels for the transitions between them are $\top$, i.e. always true, we are guaranteed that on every event we either transition into state q_0 or q_1, and verification will never be finished. The specification also asserts that if `foo` is called with 0 as the first argument, the second argument must also be 0. This is just to demonstrate how the *if* construct works.

```python
from pythonrv import rv
from pythonrv.formalrv import (formal_spec,
  make_assert, make_next)
import fibmodule

@rv.monitor(func=fibmodule.fib)
@formal_spec
def spec():
  # create assert specification
  a = make_assert(
          lambda event: event.fn.func.inputs[0] > 0)
  # create next specification
  n = make_next(spec)
  # combine by composition
  return a + n
```

Figure 5.17. A formal *pythonrv* specification function, similar to the informal specification function shown in Figure 5.2. The automata for this specification function is shown in Figure 5.18.

$$
\begin{aligned}
A(\text{spec}) &= (Q, P, q_0, S, F) \\
Q &= \{q_0, q_1, f\} \\
P &= \{(q_0, q_1, \delta(\texttt{func.inputs}[0]) > 0), (q_0, f, \neg(\delta(\texttt{func.inputs}[0]) > 0)), \\
&\quad\ (q_0, q_0, \top)\} \\
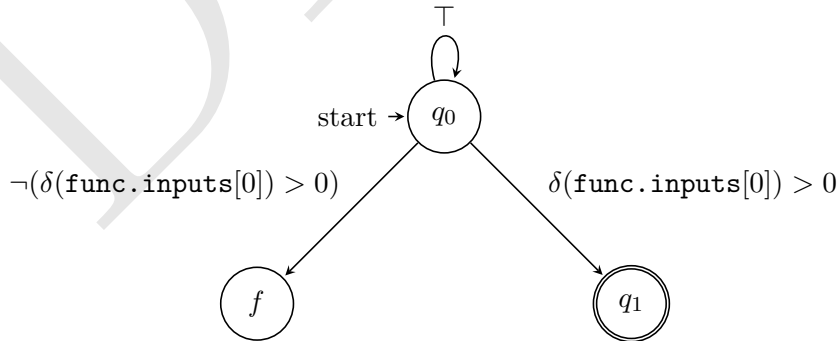S &= \{q_1\} \\
F &= \{f\}
\end{aligned}
$$

Figure 5.18. The automata for the formal *pythonrv* specification function in Figure 5.17.

```python
from pythonrv import rv
from pythonrv.formalrv import (formal_spec,
  make_assert, make_next, make_if)
import mymodule

@rv.monitor(foo=mymodule.foo,
  bar=mymodule.bar)
@formal_spec
def spec():
  # create assert specification
  a = make_assert(
          lambda event: event.fn.foo.called)

  # create if specification, with guard expression
  # and an assert specification in the then clause
  if_guard = lambda event: event.fn.foo.inputs[0] == 0
  if_then = make_assert(
          lambda event: event.fn.foo.inputs[1] == 0)
  i = make_if(if_guard, if_then)

  # create next specification
  n = make_next(spec_bar_called)
  # combine the three by composition
  return a + i + n

def spec_bar_called():
  # create assert specification
  a = make_assert(
          lambda event: event.fn.bar.called)
  # create next specification
  n = make_next(spec)
  # combine by composition
  return a + n
```

Figure 5.19. A more complex formal *pythonrv* specification function. The automata for this specification function is shown in Figure 5.20.

43

$$
\begin{aligned}
A(\text{spec}) &= (Q, P_1 \cup P_2 \cup P_3 \cup P_4, q_0, S, F) \\
Q &= \{q_0, q_1, f_1, q_2, q_3, f_2, r_0, r_1, f_r\} \\
P_1 &= \{(q_0, q_1, \lambda = \texttt{foo}), (q_0, f_1, \neg(\lambda = \texttt{foo}))\} \\
P_2 &= \{(q_0, q_2, G \wedge X), (q_0, f_2, G \wedge \neg X), (q_0, q_3, \top)\} \\
P_3 &= \{(q_0, r_0, \top), (r_0, q_0, \top)\} \\
P_4 &= \{(r_0, r_1, \lambda = \texttt{bar}), (r_0, f_r, \neg(\lambda = \texttt{bar}))\} \\
S &= \{q_1, q_2, q_3, r_1\} \\
F &= \{f_1, f_2, f_r\} \\
G &= \delta(\texttt{foo.inputs}[0]) = 0 \\
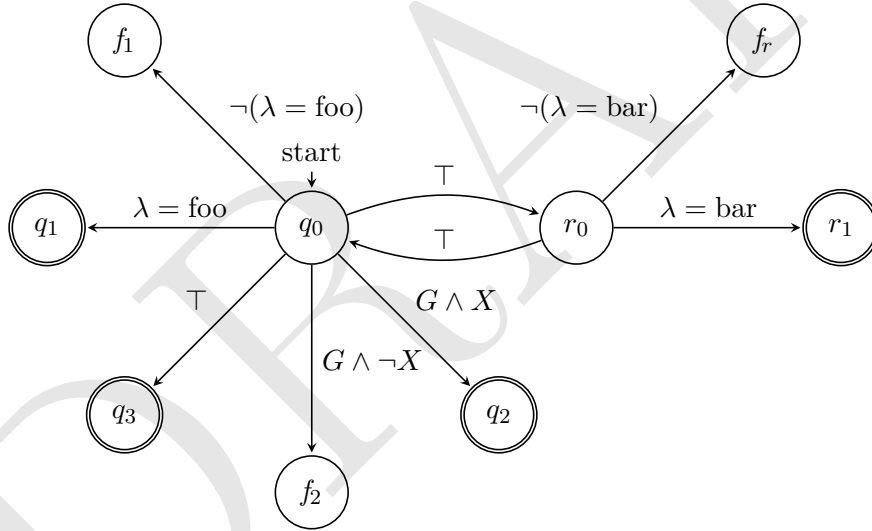X &= \delta(\texttt{foo.inputs}[1]) = 0
\end{aligned}
$$

Figure 5.20. A direct translation of the formal *pythonrv* specification function in Figure 5.19 to an automata. q_0 is the root for the `spec` specification, and r_0 is the root of the `spec_bar_called` specification. q_1 and f_1 relates to the assertion in `spec`, and q_2, q_3 and f_2 to the if statement. r_1 and f_r relates to the assertion in `spec_bar_called`.

Chapter 6

Evaluation

To see how *pythonrv* would work in a real-world setting it was incorporated into a real-time web application for Valtech Sweden, a medium-sized Swedish company.

The web application is written in Python 2.7 using the Django[1] web framework. It has approximately 10000 lines of code.

There are two questions we need to answer when writing specifications for a program. First, when, in the life-cycle of the program, should we attach the specifications? In other words, when should the code instrumentation be done? And second, and most important: what specifications should be written, and for which functions?

We can answer the first question first. It requires a bit of knowledge on the start up sequence for, and structure of, Django applications.

6.1 Technical Perspective

6.1.1 Anatomy of a Django Application

A Django application follows the Model-View-Controller pattern, or as they call it, the Model-Template-View pattern. The model is a representation of the data used by the program, and the templates are the layer that constructs the display for the user. The view links the two together, fetching the correct models for specific requests, and then delegating to the appropriate templates.

Application-specific configuration for Django programs are stored in settings modules, which are ordinary Python files. These contain settings for database connections, authentication, etc. During startup, Django reads the settings files, starts up its internal machinery, and waits for the first request.

---

[1] `https://www.djangoproject.com/`

6.1.2 When to Attach

At first glance it might seem desirable to attach the specifications before even starting the Django framework. That way we could monitor the startup process, and all of the functionality of Django.

A problem with this, that is due to how Python works, and how *pythonrv* does code instrumentation, is that *pythonrv* needs to load the modules (files) for each function to be monitored. These modules are often heavily dependent on Django, and that it has been started correctly, with all settings loaded.

A suitable time to instrument the program — to enable the specifications — is during startup, after the settings have been loaded. Some specifications, which do not monitor code dependent on the settings, could be loaded before that.

6.1.3 Technical Issues

Early in the process of using *pythonrv* in the web application it was discovered that the copying of data, such as function arguments, that *pythonrv* does would not work with Django. The latest version of Django, v1.4.1, uses a module called `cStringIO`, which produces objects that cannot be copied. All functions dealing with web requests are affected by this. This has been fixed in the development branch of Django, but in the meantime, *pythonrv* has an option to disable argument copying, either for all specifications or for a subset of them, to work around this issue.

6.2 Potential Value

Now to the most important question: what specifications could, and should, be written? What value do they provide?

The web application is only intended for internal use, for employees. Authentication of employees and authorization of their access rights are very important. Specifications can be written to make sure all views or requests are made by an authenticated user (except for the login screen), and that they are allowed to perform the requested action. A specification showing a specification verifying that proper authentication has been done is shown in Figure 6.1. This is just one way of writing a specification function for this requirement. There are several other options, such as utilizing the temporal capabilities of runtime verification with the `event.next` function. E.g., if a request asks for a resource requiring authentication, we could specify that before the response is sent, authentication must have been done.

todo: more examples Talk about where specifications would be suitable, and for what.

```python
from pythonrv import rv
from django.core.handlers.base import BaseHandler
# disable copying to work around bug in django
rv.configure(enable_copy_args=False)

# monitor the main request-processing method that
# has access to both the request and the response
@rv.monitor(bh=BaseHandler.get_response)
def ensure_auth(event):
  request = event.called_function.inputs[1]
  response = event.called_function.result
  if requires_auth(request, response):
    assert (request.user.is_authenticated(),
      "The current user is not authenticated")
    assert (request.user.is_active,
      "The current user is not active")

# a helper function for when authentication is
# required
def requires_auth(req, res):
  # only ok responses need authentication
  if res.status_code != 200:
    return False
  # requests to /login does not require auth nor
  # does /appmedia, which  is only css and images
  if (req.path.startswith("/login") or
      req.path.startswith("/appmedia")):
    return False
  return True
```

Figure 6.1. An informal *pythonrv* specification verifying that views are only accessible with proper authentication. Note how the use of Python in the specification function allows us to refactor away the condition that determines whether authentication should be required, leaving the specification much simpler and cleaner.

47

Chapter 7

Conclusions

This report, and the proof-of-concept implementation *pythonrv*, has shown that it is possible to write specifications in the target programs programming language (Python) and in a manner more similar to unit testing.

todo: excitement! and better structure

However, a few reservations should be mentioned.

The specification functions' explicit dealing with time and the actual execution flow leads to some inherent divergences from ordinary unit testing styles. This is best exemplified by the `event.next` method described in Chapter 5. Expectations could perhaps help here, see the next section.

Also, giving the specifications a formal foundation, and doing formal verification with them, is different and difficult, than with specifications already written in formal languages. The fact that the chosen programming language, Python, does not have a formal semantics defined makes the task quite a bit larger.

The formal foundation given in Section 5.5 is thus for a small subset of Python, which makes the math easier, but the resulting semantics less interesting.

If the verification parts of *pythonrv* is unwanted, it could be used as a simple framework for aspect-oriented programming. Self-healing and self-adaptation would be a very interesting use. Extracting logging functionality into separate modules, written as *pythonrv* specifications, could make the separation of concerns in the program clearer.

7.1 Future Work

The testing tool called expectations, as described in Section 4.3, could fit quite well with the *pythonrv* style of writing specifications. This could allow for a nicer way to describe temporal properties.

todo: monitor more points. @property

Especially the formal specification could do with more work on the specification language. Can this language become more like that of the informal specifications?

The performance of the implementation has not been measured or considered in

much detail. Benchmark tests for *pythonrv* would be interesting, as would attempts to introduce it as a correctness verification approach for more programs.

Offline verification, discussed in Section 2.4 and Section 5.4 would be interesting. A simple way to turn verification on or off, or to switch between online and offline, would be nice. For instance, when a bug has been found, RV could be turned on for further verification and help in finding the erroneous code.

7.2 Final Words

The trend of software systems in general seems to be toward larger and more complex entities. This makes the automated verification of program correctness, formal or not, ever more important and an essential part of software development. Runtime verification could have a place there, if it becomes more popular and simpler to integrate and use in ordinary software.

The implementation described in this report, *pythonrv*, is publicly available on the web[1] as free, open source software. People are welcome to try it, incorporate it into their programs, and extend it, as they see fit. With enough interest, *pythonrv* might develop into a mature framework for runtime verification.

---

[1] `https://github.com/tgwizard/pythonrv`

50

Bibliography

[1] D. C. Makinson, "Paradox of the preface," *Analysis*, vol. 25, pp. 205–207, 1965.

[2] J. N. Williams, "The preface paradox dissolved," *Theoria*, vol. 53, pp. 121–140, 1987.

[3] B. W. Boehm, *Software Engineering Economics.* Prentice-Hall, 1981.

[4] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, pp. 576–580,583, October 1969.

[5] R. W. Floyd, "Assigning meanigs to programs," *Proceedings of Symposium on Applied Mathematics*, vol. 19, pp. 19–32, 1967.

[6] A. Pnueli, "The temporal logic of programs," *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77)*, pp. 46–57, 1977.

[7] F. P. Brooks, Jr., *The Mythical Man-Month: Essays on Software Engineering.* Addison-Wesley, 1975.

[8] M. Leucker and C. Schallhart, "A brief account of runtime verification," *The Journal of Logic and Algebraic Programming*, vol. 78, no. 5, pp. 293–303, 2009.

[9] N. Delgado, A. Q. Gates, and S. Roach, "A taxonomy and catalog of runtime software-fault monitoring tools," *IEEE Transactions on Software Engineering*, vol. 30, pp. 859–872, December 2004.

[10] M. C. Huebscher and J. A. McCann, "A survey of autonomic computing degrees, models, and applications," *ACM Computing Surveys*, vol. 40, pp. 7:1–7:28, August 2008.

[11] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *Proceedings of the 21st international conference on Software engineering*, ICSE '99, (New York, NY, USA), pp. 411–420, ACM, 1999.

[12] A. Bauer, M. Leucker, and C. Schallhart, "Monitoring of real-time properties," in *In Proceedings of the 26th Conference on Foundations of Software Technology*

and Theoretical Computer Science (FSTTCS), volume 4337 of LNCS, pp. 260–272, Springer, 2006.

[13] H. Barringer, A. Goldberg, K. Havelund, and K. Sen, "Rule-based runtime verification," 2003.

[14] H. Barringer, D. Rydeheard, and K. Havelund, "K.: Rule systems for run-time monitoring: From eagle to ruler," in *In: RV'07: 7 th International Workshop on Runtime Verification. Revised Selected Papers*, pp. 111–125, 2007.

[15] B. Meyer, "Applying "design by contract"," *Computer (IEEE)*, vol. 25, pp. 40–51, October 1992.

[16] D. S. Rosenblum, "A practical approach to programming with assertions," *IEEE Transactions on Software Engineering*, vol. 21, pp. 19–31, January 1995.

[17] E. Bodden, "Efficient and Expressive Runtime Verification for Java," in *Grand Finals of the ACM Student Research Competition 2005*, March 2005.

[18] E. Bodden, "A lightweight LTL runtime verification tool for Java," in *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28, 2004, Vancouver, BC, Canada*, pp. 306–307, ACM, 2004. ACM Student Research Competition.

[19] D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim, "Jass – java with assertions," *Electronic Notes in Theoretical Computer Science*, vol. 55, no. 2, pp. 103–117, 2001. RV'2001, Runtime Verification (in connection with CAV '01).

[20] S. Jalili and M. MirzaAghaei, "Rverl: Run-time verification of real-time and reactive programs using event-based real-time logic approach," in *Proceedings of the 5th ACIS International Conference on Software Engineering Research, Management & Applications*, SERA '07, (Washington, DC, USA), pp. 550–557, IEEE Computer Society, 2007.

[21] K. Beck, "Simple smalltalk testing: With patterns." http://www.xprogramming.com/testfram.htm, Retrieved on 2012-07-03.

[22] M. Fowler, "Xunit." http://www.martinfowler.com/bliki/Xunit.html, Retrieved on 2012-07-03.

[23] M. Fowler, 2007. http://martinfowler.com/articles/mocksArentStubs.html, Retrieved on 2012-08-22.

Appendix A

Glossary

This appendix consists of a small glossary with short explanations of a few key concepts used in this report, and pointers to where they are described in more detail.

incompleteness As in the incompleteness of testing, reflects the fact that, with testing, not all possible states are examined. Some formal methods are complete.

formal methods Techniques for mathematical reasoning about program correctness. See Section 2.2.

linear temporal logic, LTL See Section 3.2.1.

mock, mocking Using fake, stand-in objects to isolate units of the program from the rest of the system. See Section 4.2.

model, system model A conceptual model and abstraction of a system. See Section 2.1.

model checking See Section 2.2.

runtime verification Verifying an execution of a program. See Section 2.4 for an overview and Chapter 3 for a more in-depth description.

self-healing, self-adapting The concept of systems that can analyse itself, react to events, and enact modifications to fix or circumvent errors, and possibly add improvements to existing functionality. See e.g. [10] for more.

specification Something that describes the correct behaviour of something else. See Section 3.1.

state explosion problem Concerns the problem of the exponential increase in the state-space when more variables of a system are taken into consideration in the system model.

testing An approach for program verification. See Section 2.3.

undecidability A (decision) problem is undecidable if it is impossible to construct an algorithm for it that always gives the correct answer. One example of an undecidable problem is the *halting problem*.

unit testing Dividing the program into small units, testing each separately. See Chapter 4.

verification Checking the correctness of a program, by using techniques such as testing or formal methods. See Section 2.1 for a more thorough definition.