KTH Computer Science
and Communication

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

TRITA xxx yyyy-nn

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 useful 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 the 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 relatively simple. It also shows that formal specifications can be written in Python, but in a more unwieldy syntax and structure than the informal one. Many interesting properties can be verified using it that ordinary tests would have trouble with.

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

Mjukvarusystem växer sig allt mer komplexa, och mer sofistikerade verktyg krävs för att säkerställa att system fungerar korrekt — att de opererar enligt sina specifikationer. Traditionella tillvägagångssätt för programverifiering tillför mycket, men de har också sina nackdelar. Formella metoder kan ge användbara matematiska bevis om korrektheten av program, men de är komplexa och är svåra att använda. Ofta opererar de bara på en konstruerad systemmodell, inte det faktiska programmet. Testning, som metod, har å andra sidan en enkel syntax och bra verktygsstöd, och används i stor utsträckning. Men testning är informell och ofullständig, och testar bara de specifika testfall som testkrivarna kan komma på.

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

Avsikten här har varit att undersöka hur testning, och specifikt enhetstestning, kan kombineras med runtime-verifiering. Detta genom att visa 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 möjliggör en effektiv användning av runtime-verifiering. Den är enkel att integrera i existerande program, och den informella specifikationssyntaxen är förhållandevis enkel. Den visar också att formella specifikationer kan skrivas i Python, men i en mer omständigt syntax och struktur än den informella. Många intressanta egenskaper, som vanliga tester skulle ha problem med, kan verifieras med denna metod.

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 have 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 would 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 5.

And last, I would 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. In this report, we investigate how the syntax and tooling of unit testing can 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 Contribution

In this report we give a background on runtime verification and unit testing, and describe a proof-of-concept implementation called *pythonrv*. *pythonrv* is a runtime verification framework for Python. It allows the specification writer to write specifications, both formal and informal, directly in Python. It makes good use of the Python decorator syntax, and gives a clean separation between the program and the specifications, while allowing the specifications to use the power of Python. It is the approach to formal verification for specifications written in Python that is our main contribution.

Naturally, there are some areas that are left out-of-scope for this report. We give a formalization for only a subset of Python — for specifications constructed in a special way. We do not attempt to formalize Python.

Also, we only deal with a single thread of execution — no multi-threading or multiple processes — and we do not handle real-time specifications. No performance testing or benchmarking is done on the implementation.

These limitations to this work are of course all very interesting, and would be very suitable for future work in this area.

1.4 Report Organization

In this chapter we have given a short introduction to and description of the problem matter. In the next two chapters we introduce runtime verification and unit testing, see Chapters 2 and 3, respectively. In these chapters we give a background to the area of program verification, and delve into the details of specification languages, approaches to code instrumentation, and current ideas in unit testing.

In Chapter 4 we describe the approach we take in this work to solve the problem stated in Section 1.1. We describe the syntax, instrumentation and verification techniques used in a proof-of-concept implementation, and give a formal foundation to a subset of the syntax. In Chapter 5 we then give an evaluation of applying this work on a real-life web application.

Finally, in Chapter 6 we draw some conclusions and discuss this and future work.

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.

Chapter 2

Introduction to Runtime Verification

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 we start this chapter in Section 2.1 by giving definitions for some concepts used in this report, before we delve into some background on formal methods in Section 2.2. We give an overview of runtime verification in Section 2.3.

Runtime verification is a technique for verifying a program's compliance against a specification during runtime. These specifications need to be written somehow, which we discuss in Section 2.4. Approaches for verification are discussed in Section 2.5. 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. We discuss this in Section 2.6.

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 specifications, 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.

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 for reasoning about program correctness. 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 attempts to solve this by dealing directly with the system, creating its model at runtime.

2.3 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 the previous section. Runtime verification aspires to be a light-weight formal verification technique, see e.g. [7, 8]. 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 4). 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. [9]).

Leucker et al. present a definition of RV in [7], together with an exposition of the advantages and disadvantages, similarities and differences, with other verification

approaches. In [8], Delgado et al. classify and review several different approaches to and frameworks for runtime verification.

The next chapter examines RV in more detail.

2.4 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 reference implementations[1], to tests (see e.g. Chapter 3), and to *formal specifications*, written in formal languages, specifying properties that should verifiably hold for the program. They are is a mathematical constructs that can be used in verification proofs to show that a program works correctly.

It is these last two types of specifications, tests and formal specifications, that we are interested in in this report, and which play an important role in our approach to 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.

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, in test suites, etc.

Runtime verification requires that the specifications are accessible when building and running the program. 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.

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.

Other specification languages take a more informal approach, some allowing arbitrary code as part of the specification. Below we introduce and give an overview to the most important and common specification languages.

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

2.4.1 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 [10], 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 2.1.

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

Figure 2.1. 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. [11] 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 are counterparts to LTL in the real-time setting such Timed Linear Temporal Logic (TLTL) and Metric Temporal Logic (MTL). They introduces clocks and time constraints to make specifications of real-time properties possible. The concept of real-time specifications is of great interest to runtime verification, but will not be discussed further here as it is out of scope for this work. See e.g. [11, 12] for more.

Several runtime verification approaches use LTL or versions thereof as specification language, see e.g. Bauer et al. [11], Bodden [13], Kähkönen et al. [14], Java PathExplorer by Havelund et al. [15], Temporal Rover by Drusinsky [12].

2.4.2 Eagle and RuleR

Barringer et al. have written EAGLE [16] and RULER [17] 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 2.2. It describes the LTL operator $\boldsymbol{G}$. $\bigcirc$ 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 2.2. A simple example of an EAGLE formula, semantically describing the LTL $\boldsymbol{G}$ operator. Taken from [16].

The same property can be described in RULER. If we let $\{r\}$ be the initial set of rule activations, the formula in Figure 2.3 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 2.3. A simple example of a RULER formula, semantically describing the LTL $\boldsymbol{G}$ operator. Taken from [17].

An EAGLE formula describing the LTL formula from Figure 2.1 is shown in Figure 2.4. 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.

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

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

Figure 2.4. A simple example of an EAGLE formula, semantically the same as the LTL formula from Figure 2.1.

Both EAGLE and RULER are strictly more powerful than LTL; e.g., they both support specifications that allow matching function calls with function returns, which is in the realm of context-free grammars. LTL does not support this [16, 17].

2.4.3 Design by Contract

Design by Contract is a specification language for runtime verification without the ability to describe temporal properties. Design by Contract was introduced by Bertrand Meyer in [18], and has been fully implemented in the Eiffel programming language[2].

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

```
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 2.5. An example showing the use of the Design-by-Contract concepts of pre- and post-conditions, written in Eiffel. Example taken from [18].

An example of a contract is shown in Figure 2.5. This example shows a method `put-child` for adding nodes to a tree-structure. Pre-conditions are written in the `require` block, post-conditions in the `ensure` block, and the function body in the `do` block. The contract in the example says that if we provide an element, it will be inserted as a child to the `Current` node, which will thus have one more child. Note the use of the language construct `old` to get the value of an attribute at before the function executed.

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[3] in .Net, allow for arbitrary code in the specifications. The specifications are thus informal, unless a formalization is given

[2] `http://eiffel.com/`

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

for the programming language used. The Design by Contract specifications used in Jass (Java with assertions) have formal semantics [19].

2.4.4 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.

```
#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 2.6. An example showing the use of `assert` in C.

An example of the use of assertions in C is shown in Figure 2.6. The example function takes an array of characters and its size and returns a copy of it. Assertions are used as pre-conditions to make sure that the inputs are valid — non-null and non-negative. Note that the return value of `malloc` is checked to be non-null using an if-statement — this is a case that could happen (if the system runs out of memory), and should thus be handled by proper error-handling logic. 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.

2.5 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 for formal verification, see e.g. [11, 13, 20, 16], 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 examined 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.

Another approach for verification is to rewrite the instrumented code, adding assertions that check the properties of the specifications, see e.g. [21, 12]. This might be difficult to do with more complex specifications.

2.6 Code Instrumentation

For verification to work, the verifier needs access to events happening in the program. Depending on the system model of the specification language, such events can be function calls, statement executions, variable assignment, etc. 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.

In the following sections the main approaches for program instrumentation are discussed.

2.6.1 Pre-processing the Code

Rosenblum [21] and Drusinsky [12] use a pre-processor step in the compilation setup to instrument code, where the specifications (called assertions by Rosenblum) are transformed from comments into regular code. The verification code is then compiled together with the program.

An example is shown in Figure 2.7. In this case, `assume` work as pre-conditions and `promise` as post-conditions, similar to contracts in Design by Contract. `in x` means "the value of `x` upon entering the function".

2.6.2 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-

```
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 2.7. An example showing instrumentation via special comments and code pre-processing. Example taken from [21].

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

2.6.3 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. See e.g. [22] for approaches using dynamic code rewriting.

2.6.4 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[4] is the canonical framework for aspect-oriented programming.

Runtime verification is a typical case of a cross-cutting concern. Bodden [13] and Kähkönen et al. [14] use AspectJ in their runtime verification implementations. The specifications are however written adjacent to the code or interfaces being monitored, as Java Annotations.

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

Inserting aspects into programs is called *aspect weaving*. It can be implemented as a pre-processing or post-processing step, or dynamically during runtime or when code is loaded [23, 22].

2.6.5 No instrumentation

Some runtime verification specifications can be written and executed without the need for extra instrumentation of the system. Such specifications could monitor the surrounding environment, e.g. check availability of required services, or make sure that enough disk space is free. Other approaches could be to use existing features of the system, such as its logging, for runtime verification. The logs can be parsed by a separate process, during the execution or as a post-mortem, do determine that the execution followed the specification. Barringer et al. do this in [24].

Chapter 3

Introduction to Unit Testing

An approach to verification quite different from formal methods 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. [25]).

3.1 Unit Testing

Unit testing is the concept of writing tests for small units in a program, such as functions, classes, etc. The separate parts — silos, with minimum dependencies to the rest of the system — are tested in isolation during development to test their functionality. The aim is to reduce the risk of breaking existing functionality when developing new features, or modifying existing code, by preventing regression. Dividing a program into units can be difficult, and proponents of unit testing argue that it encourages 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

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

is very old.

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.

3.2 xUnit

Kent Beck introduced the style of the modern unit testing framework in his work on a testing framework for Smalltalk [26]. 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 [27]. 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.

The xUnit style of unit testing [27] 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, shown in Figure 3.1. Jasmine, being younger and more "hip", has a slightly different syntax, but essentially the same structure; see Figure 3.2.

In JUnit, and xUnit, you run a *test suite* of *test cases*, which contain tests. The example in Figure 3.1, 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 3.1 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 3.1 are traditional unit tests.

3.3 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

[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/`

```
// 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 3.1. 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.

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.

```
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 3.2. 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.

3.4 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* [28]. A simple example of expectations is shown in Figure 3.3.

Figure 3.3 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.
- 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 3.3 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.

```
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 3.3. An example of expectations, written using jMock and JUnit. Example taken from [28].

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/

Chapter 4

Approach

In this report we 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 this end, there are four issues we address:

Syntax We define the syntax and general structure of specification function in the next section.

Instrumentation How these specifications gain access to the target system is described in Section 4.2.

Formalization A subset of python, using a special structure and methods, is given a formal foundation, and specifications written are considered formal. This is described in Section 4.4.

Verification We have chosen to perform verification online and synchronous with the target system. The technique for verification differs for informal and formal specification functions. See Sections 4.3 and 4.4, respectively.

This chapter is a documentation on how we have chosen 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].

During the development of this proof-of-concept, the biggest factor in deciding what language to use was how it would assist in instrumentation. 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 5.

Python[2], among several languages, fits these criteria, and was chosen as the implementation language. Python is a dynamic programming language, usually running inside a virtual machine. Perhaps the most striking feature of Python is

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

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

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 of the *pythonrv* specifications are naturally heavily influenced by the fact that they are written in Python.

pythonrv is not the first runtime verification framework for Python, see e.g. LogScope [24]. But it is unique in its use of pure Python as specification language.

4.1 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 3.

4.1.1 General Structure of a Specification

Specifications in *pythonrv* are structured as *specification functions*.

Definition 1. In *pythonrv*, a *specification function* is a Python function describing a specification, which *pythonrv* can use for verification of the program. A specification function takes a special *event* argument. A specification *monitors* points of the program, and the points being monitored are called *monitorees*. *pythonrv* supports only monitoring of function calls.

The restriction that *pythonrv* only support monitoring of function calls is not a very limiting one. For example, in Python, operators such as + and − are functions, and reading and writing variables can be turned into function calls through the *property* construct and other so-called magic functions. See the Python documentation[3] for more details. *pythonrv* thus supports monitoring of these events as well.

A specification function consists of any valid Python code. It is passed the special `event` during verification, which gives the specification function access to data about the current event.

In Figure 4.1 we show what happens when a monitored function is called. The monitoree is wrapped, so the function call first reaches the verification code. This will do some setup and pre-processing and then pass the event on to the specifications monitoring this function. If these do not find any violations, the actual function is called, and the return value is sent back to the system. As we'll see later in this section and in Section 4.2, this is a somewhat simplified version. Verification can, at the discretion of the specification writer, occur after the function call.

[3]Specifically `http://docs.python.org/2/reference/datamodel.html`.

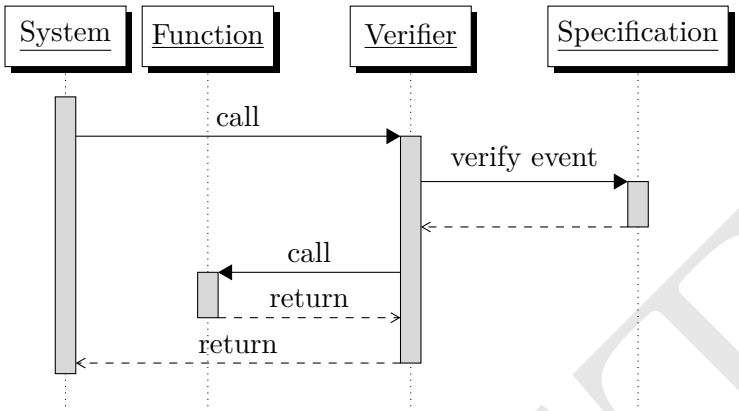


Figure 4.1. The sequence for verifying a function call event.

The structure of a informal *pythonrv* specification function is shown in pseudo-code in Figure 4.2, and the structure of the special `event` parameter is shown in Figures 4.3, 4.4 and 4.5.

Note that the Python `assert` statement raises an `AssertionError` exception when its argument evaluates to `False`. This is used in *pythonrv* to represent a specification violation, see Section 4.3.

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

decorate the spec with its monitorees
define specification function spec(event):
  # perform verification here
  if bad input:
    raise an AssertionError exception

  assert properties that should be true
```

Figure 4.2. The structure of a *pythonrv* specification.

```
# rv.py
# the Event class has some special
# functions
class Event:
  def finish(success):
    if success:
      self.success()
    else:
      self.failure()
  def success():
    # remove this spec from future
    # verification
  def failure():
    # fail the spec by raising an
    # exception
    raise AssertionError("Violation")

  def next(new_spec):
    # add new_spec to be used in
    # verification - on the next
    # event only
  def next_called_shoud_be(monitoree):
    # on the next Event, the function
    # called should be monitoree. Verify
    # this
```

Figure 4.3. The structure of the special *event* argument passed to *pythonrv* specification functions. The example is written in pseudo-python.

```
# rv.py, cont'd
# an Event object is populated with data
event = Event()
# list of previous events
event.history
# the event that occurred before this
event.prev = history[-1]

# map (name => FunctionCall) for functions
# monitored by the spec
event.fn
# the monitored function that generated
# this event
event.called_function
```

Figure 4.4. The structure of the data in the *event* argument passed to *pythonrv* specification functions. The example is written in pseudo-python.

```
# rv.py, cont'd
# the FunctionCall class
class FunctionCall:
  def next(new_spec):
    # same as for Event but only for Events
    # where this function was called

fc = FunctionCall()
# the name of the function
fc.name
# True if the function was called in this event
fc.called
# the input arguments for the function
fc.inputs
# the output arguments for the function
fc.outputs
# the function's return value
fc.result
```

Figure 4.5. The structure of the function call structures in the *event* argument passed to *pythonrv* specification functions. The example is written in pseudo-python.

4.1.2 Specification Functions by Example

```
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 4.6. A very simple *pythonrv* specification function.

We now proceed with a few simple examples to show how informal *pythonrv* specifications can be written. More realistic examples are shown in Chapter 5.

The first example, in Figure 4.6, shows the basics of a *pythonrv* specification function. The specification just verifies, through the `assert` statement, that the first input to the monitored function is always greater than zero.

On line 1 we import the `rv` module from the `pythonrv` package, which gives us access to the runtime verification features of *pythonrv*. We specify that the specification should monitor the function `fib` in the module `fibmodule` on line 4, and that whenever `fib` is called, the specification should be verified. The monitoree `fib` is, locally to the specification function, aliased as `func`. The instrumentation is also done on line 4 by using the *function decorator* `rv.monitor` (see Section 4.2 for an explanation of function decorators and `rv.monitor`.)

On line 5 we define the specification as an ordinary Python function called `spec`, taking one argument, `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 4.6 will be called upon every invocation to `fibmodule.fib`.

In the second example, Figure 4.7, we show how a specification function can monitor two functions. The specification function will be called whenever either of the monitored functions are called. The example specification verifies that the calls to the two functions alternate; that no two calls to `foo` occur without a call to `bar` in between, and vice versa. The first call has to be to `foo`.

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.

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

In the last example, Figure 4.8, we show a more advanced specification, in which the `next` function of the `event` argument is used. `event.next` allows the

```
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 4.7. A *pythonrv* specification that monitors two functions, `mymodule.foo` and `mymodule.bar`.

specification function to add more specification functions (possibly implemented as closures or lambdas) to be executed when the next event occurs.

The specification function 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 `baz`. After that, any calls are allowed — the specification function will not be used in verification any longer. So, the call event sequence (`foo()`, `baz(True)`, `baz(True)`, `baz(True)`, `bar()`) is accepted, but `baz(True)`, `bar()` is not.

The specification is split into two parts, `spec` and `followup`. `spec` verifies that `foo` is called first, and then that `followup` should be verified on the following event. After that, on line 11, `spec` notifies that it has finished its verification, and should not be used in the future — it "unsubscribes" from future events.

On line 10 the function `followup` is added to be executed on the next event. We use `followup` to accept any calls to `baz` (with the first argument set to `True`). `followup` finishes its verification when `bar` has been called.

Since `followup` is added to the verification machinery through the `event.next` function — as a "oneshot" specification function — it needs to add itself using `next` for verification on subsequent events. We do this on line 24.

4.1.3 Customization

The examples above show the main capabilities of *pythonrv* specifications. However, some details can be customized through the `@rv.spec` decorator, which accepts the following parameters (default values in parentheses):

```
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 4.8. A more complex example, utilizing the `event.next` function.

`when` (`rv.PRE`). This argument allows the specification writer to determine when the specification should be verified, before or after the monitorees are called. The valid arguments `rv.PRE` and `rv.POST`, respectively. Only when `when=rv.POST` does the specification function have access to the monitorees' return values.

`history_size` (2). Specifies how much event history should be remembered for the specification. `rv.INFINITE_HISTORY_SIZE` can be used to remember all previous events, although this could naturally be a big drain on the available memory resources.

`level` (`rv.ERROR`). Specifies how severe or important the specification is. Any value can be used; `DEBUG`, `INFO`, `WARNING`, `ERROR`, `CRITICAL` are supplied by `pythonrv`. With these levels we can have more control over what should happen when a specification violation occurs, see Section 4.3.

4.2 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 a function as a parameter, possibly modifies it, or uses it in some way (decorates it), and then returns it (or another function). This is used throughout Python to, for instance, turn functions into static or class methods. In Figure 4.9 we show an example function decorator definition, and in Figure 4.10 we show the syntax for using it.

```
1   # function_decorator.py
2   # the function decorator
3   def cache(decoratee):
4     decoratee.cache = dict()
5     # define the closure ("inner function")
6     def wrapper(x):
7       if not x in decoratee.cache:
8         # call the decorated function
9         decoratee.cache[x] = decoratee(x)
10      return decoratee.cache[x]
11    return wrapper
```

Figure 4.9. An example of how to define a function decorator.

The decorator in Figure 4.9 caches the return values for functions on which it is applied, so that future calls to a decorated function with the same arguments will short-circuit and return the cached value directly, never entering the function itself. As is shown in Figure 4.10, the decorator is something that can be attached with a special syntax, using the `@` operator. The decorated function does not need any further modifications.

`rv.monitor` is a decorator similar to the `cache` decorator above. It takes to set of arguments, first what functions should be monitored (the monitorees), and second the specification function. However, the `rv.monitor` decorator does not modify the function which it decorates, which here is the specification function, but instead finds the monitorees and decorates them with a dynamically generated function `wrapper`. The code in Figure 4.11 illustrates how *pythonrv* does instrumentation.

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, 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

```
1   # test.py
2   from function_decorator import cache
3
4   def fact(x):
5     if x <= 1:
6       return 1
7     return x*fact(x-1)
8
9   print fact(5) # >> 120 (5 calls to fact)
10  print fact(5) # >> 120 (5 calls to fact)
11
12  # decorate fact2
13  # equivalent to fact2 = cache(fact)
14  @cache
15  def fact2(x):
16    if x <= 1:
17      return 1
18    return x*fact2(x-1)
19
20  print fact2(5) # >> 120 (5 calls to fact2, results cached)
21  print fact2(5) # >> 120 (0 calls to fact2)
```

Figure 4.10. An example of how to use the function decorator from Figure 4.9.

the monitored function.

Usually in Python, functions belong to a parent *container*, such as a class, an object, or a module[4]. In Figure 4.10 the functions `fact` and `fact2` 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).

So in the *pythonrv* instrumentation process, the parent container of each monitoree is extracted, and the reference to the monitoree is overwritten with a reference to the wrapper. The wrapper still has a reference to the monitoree.

The implementation of the instrumentation code in *pythonrv* is more optimized than what is shown in Figure 4.11. For instance, each monitoree only ever have one wrapper function, independent of how many specifications want to monitor it. The wrappers then iterate over the relevant specifications, passes the control (and an event) to them, and then runs the monitoree. As we stated in section 4.1, a specification can either be verified before or after the function call. The wrapper function handles this.

[4]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   # 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)
18
19  monitor(myfunc, myspec)
20  myfunc(x, y, z) # will call myspec first
```

Figure 4.11. 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.

4.3 Verification of Informal Specifications

In *pythonrv*, verification of informal specifications means just executing them. The specification functions are valid python functions, and executing them on the appropriate events, providing access to the corresponding data, verifies the specification they represent.

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 4.1).

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, re-raises the `AssertionError` exception, and thus causes it to propagate up through the call stack. If the exception isn't caught and suppressed by code higher up the call stack, the program will crash. `AssertionError` are supposed to be raised when something has gone wrong and the operation should halt, so it is bad coding practice to suppress them.

A second error handler just logs the error message, using the standard Python logging module.

We can customize what should happen when verification fails for a specification. We do this by creating an error handler and passing it to the `rv.configure` function. Any object with a `handle` method that takes two arguments, the specification `level` and a list of errors, is a valid error handler.

In Figure 4.12 we use a simple error handler that only propagates the verification errors, and thus crashes the program, for specifications with level `rv.CRITICAL`. A specification with `level` set to `rv.CRITICAL` is also shown.

```
1   from pythonrv import rv
2   class CustomErrorHandler(object):
3       def handle(self, level, errors):
4           if level == rv.CRITICAL:
5               # raise only the first error
6               raise errors[0]
7
8   rv.configure(error_handler=CustomErrorHandler())
9
10  @rv.monitor(...)
11  @rv.spec(level=rv.CRITICAL)
12  def spec(event):
13      assert False
```

Figure 4.12. How to use a custom error handler.

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. This is suggested as an area for future work in Chapter 6.

4.4 Formal Foundation

The purpose of a formal foundation for a verification approach is so that we can reason mathematically about the specifications. To this end, 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 4.4.3

A seemingly insurmountable problem quickly arises when attempting to give such a mathematical representation to the *pythonrv* specifications described in Section 4.1. *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[5].

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, which we describe in the next section.

These formal specifications can be represented as automata. The verification approach takes a specification automata and combines it with the system model (described in Section 4.4.3), and the resulting product (a tree) can be traversed to see if the system model of the program satisfies the specification.

4.4.1 Python Subset for Formal Specification Functions

The subset of Python that we can use in *pythonrv* for formal verification allows us to write *formal-specification-functions*.

Definition 2. A *formal specification function* is one of four basic function structures: The *assert*, *next*, *if-then* and *if-then-else* functions, shown in Figure 4.13. The expression E used in the *assert*, *if-then* and *if-then-else* functions (lines 2, 14 and 21 in Figure 4.13) denotes any idempotent, immutable, valid Python boolean expression (a boolean expression is one which evaluates to either true or false). 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`.

These formal specification functions can be composed together to form more complex specifications. Composition is done at *composition points*, using the *composition operator* $\circ$. We describe composition in Section 4.4.2.

Definition 3. 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*.

Definition 4. Composition is described by the *composition operator* $\circ$. 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).

[5]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.

```
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_then():
13    then = # required composition point 'then'
14    i = make_if(E, then)
15    tail = # optional composition point 'tail'
16    return i + tail
17
18  def if_then_else():
19    then = # required composition point 'then'
20    els = # required composition point 'else'
21    i = make_if(E, then, els)
22    tail = # optional composition point 'tail'
23    return i + tail
```

Figure 4.13. The four basic formal specification functions.

Definition 5. 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. Only the *assert* basic specification function is complete — the other can become complete through composition.

A complete and verifiable formal specification function also needs some boilerplate, shown in Figure 4.14.

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-then* or *if-then-else* 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.

The four basic formal specification functions correspond to simple, deterministic, finite automata, sketches of which are depicted in Figure 4.15. Please note that the automata in Figure 4.15 are only sketches (as can be seen by the squiggly cloud-shapes), and are only supposed to give an overview of how they work. They are defined properly in the next section.

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

Figure 4.14. The boilerplate code for formal *pythonrv* specification functions that allow for verification and instrumentation to take place.

Definition 6. Every formal specification function s can be represented by a deterministic 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 *transition label*. An automata has one initial state q_0, a set of *success states* S and a set of *fail states* F. Note that the composition of automata might make them non-deterministic, i.e. more than one transition with the same label. This is dealt with in the verification, which is fully deterministic.

Definition 7. A state cannot be both a success state and a fail state — it 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.

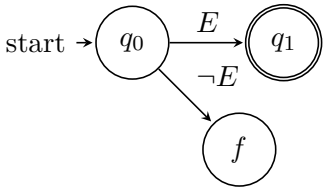
Definition 8. Each transition in the automata has a label — an expression — which is described in detail in Section 4.4.3. The label is taken from the expressions in the Python functions. The label $\top$ denotes "any and all expressions".

Definition 9. The set of *initial transitions* T, $(q_0, q', E) \in T$, $T \subseteq P$ are especially important for composition, see Section 4.4.2. All initial transitions start from the initial state q_0, go to some state q', and have some label E. When we describe an automata we will usually split up its transitions P into two parts, the initial transitions T, and all other transitions R: $P = T \cup R$ and $T \cap R = \emptyset$.

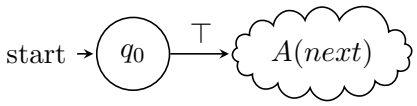
These preliminaries allow us to define each basic formal specification function mathematically as automata, and formal rules on how to compose them.

4.4.2 Automata Representation and Rules for Composition

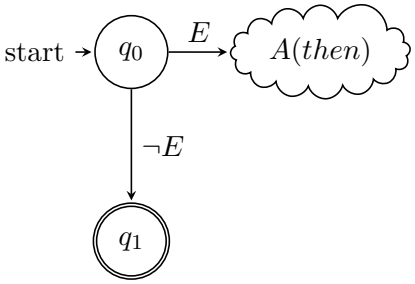
With each formal specification function represented as an automaton, we can use composition to build more complex and interesting specifications. Composition is defined inductively, preserving the semantics, which we define in Section 4.4.3.



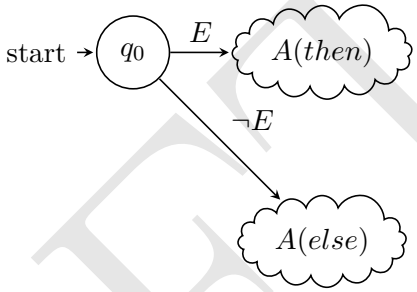
(a) A generalized *assert* specification, asserting the expression E.



(b) A template for a *next* specification.



(c) A template for an *if-then* specification.



(d) A template for an *if-then-else* specification.

Figure 4.15. Sketches of automata for the four basic formal specification functions. The squiggles around $A(next)$, $A(then)$ and $A(else)$ denote that some composition is required there.

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 4.15 (a), is defined as:

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

If $\neg E$ 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 functions and compositions thereof, at least one open composition point, the *tail* composition 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 transitions 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:

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

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 4.16. The resulting automata is unnecessarily complex, and can be simplified to a smaller automata with the same semantics, as seen in Figure 4.17.

$$\begin{aligned} A(a_E) &= (\{q_x, q_y, f\}, \{(q_x, q_y, E), (q_x, f, \neg E)\}, q_x, \{q_y\}, \{f\}) \\ A(a'_{E'}) &= (\{q_z, q_w, f'\}, \{(q_z, q_w, E'), (q_z, f', \neg E')\}, q_z, \{q_w\}, \{f'\}) \end{aligned}$$

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

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

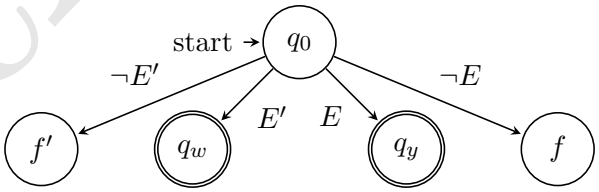


Figure 4.16. An example showing a composition of an *assert* specification a_E with another *assert* specification $a'_{E'}$. Only the resulting $A(a_E \circ_{tail} a'_{E'})$ automata is shown. q_y and q_w are success states; f and f' are fail states.

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 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:

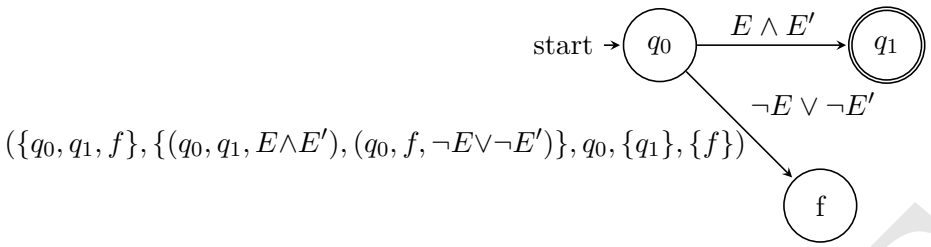


Figure 4.17. A simplified version of the automata from Figure 4.16, semantically identical. The success states have been merged into q_1, and the fail states into f.

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

This is illustrated in Figure 4.18.

Also note that composition using the *next* composition point is associative, as shown in Figure 4.19.

$s' = n \circ_{next} s$

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

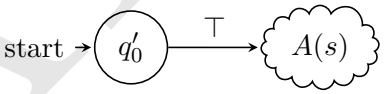


Figure 4.18. The composition of a *next* specification with any specification s, using the *next* composition point.

$$\begin{aligned} (n \circ_{next} s_1) \circ_{tail} (n' \circ_{next} s_2) &= n \circ_{next} t \\ t &= s_1 \circ_{tail} s_2 \end{aligned}$$

```
def s():
  return make_next(s1) +
    make_next(s2)

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

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

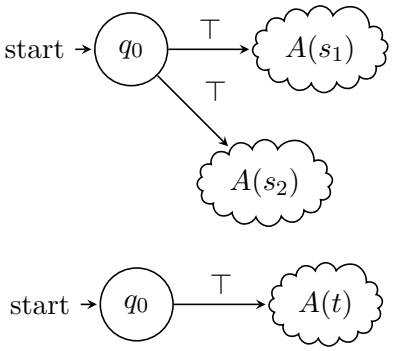


Figure 4.19. The associativity of the *next* composition point in a *next* formal specification function.

The *if-then* and *if-then-else* specifications

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

In an *if-then-else* specification i_E we consider the expression E as a guard for the *then* composition point, and $\neg E$ as a guard for the *else* composition point. In *if-then* specifications, the $\neg E$ guard goes to a success state.

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 $\neg E$ to the labels of all initial transitions for the automata at the *else* composition point. The guards, E and $\neg E$ are added as parts of a conjunctive.

Let s_1 be the specification attached to the *then* composition point and s_2 attached to the *else* composition point, and:

$$\begin{aligned} A(s_1) &= (Q_1, T_1 \cup R_1, q_{10}, S_1, F_1) \\ A(s_2) &= (Q_2, T_2 \cup R_2, q_{20}, S_2, F_2) \end{aligned}$$

For *if-then* specifications compositions becomes:

$$\begin{aligned} A(i_E \circ_{then} s_1) &= (Q, P, q_0, S, F) \\ Q &= Q_1 \cup \{q_0, q_1\} - \{q_{10}\} \\ P &= T_1' \cup R_1 \cup X \\ T_1' &= \{(q_0, q, E \wedge E') \mid (q_{10}, q, E') \in T_1\} \\ X &= \{(q_0, q_1, \neg E)\} \\ S &= S_1 \cup \{q_1\} \\ F &= F_1 \end{aligned}$$

And for *if-then-else* specifications compositions becomes:

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

4.4.3 Semantics

The semantics for the formal specifications described above is quite straightforward. The automata representations and composition rules preserve the semantics of the

Python code, and together with a description of a system model we can deduce whether a specification would accept or reject an instance of such a model.

Generating a System Model-Specification Product

The system model in *pythonrv* is quite simple.

Definition 10. The *system model* 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 — even a list), where all nodes but the last have one edge, leading to the next event. The first edge is labeled α_0, the second is labeled α_1, etc. See Figure 4.20 (a) for an example.

In the same way that the system model can be viewed as a graph, so can a specification automata, see Figure 4.20 (b). All graphs have a depth.

Definition 11. The *depth* of a graph rooted at a node q is the number of transitions required to reach the node farthest away from the q. The depth of a graph containing cycles is considered infinite.

Definition 12. 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 4.20 (c). The product Z will be a tree of depth $min(depth(M), depth(C))$, and it is created through the following recursive procedure:

1. Let M be a system model rooted at s_0. Let C be a specification automata rooted at q_0.

2. Let (q_0, s_0) be the root node of our product graph.

3. Merge the initial transitions in C (those starting from q_0) with the single transition in M starting from s_0, (s_0, s_1, α_0). This yields a set of transitions T:

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

   The labels of the resulting transitions are of the form $\alpha \models E$, where α is the label of the system model transition, and E is the label of the specification transition. See below for what this means.

4. Let Q be the those nodes reachable in T from (q_0, s_0):

$$Q \quad = \quad \{(q,s) | (r,(q,s),\alpha \models E) \in T\}$$

5. Recursively create the transitions from the nodes in $(q,s) \in Q$ by starting from step 2. Bottom out when there are no more transitions leading out of q or s, i.e. the specification automata's or the system model's depth have been reached.

A state (q,s) of the product Z, where q is a state from the specification and s is from the system model, is a success state if q is a success state, and a fail state if q is a fail state.

In Figure 4.20 we show an example of a system model, a specification, and the resulting product of combining them.

Verification

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):

$$\alpha = (\lambda, \delta) \quad \models \quad \text{expression } E \text{ over } \alpha(\lambda, \delta)$$

The expression E is translated directly from Python expressions in the specification function. An incomplete sketch of the translation procedure, shown in Figure 4.21, 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.

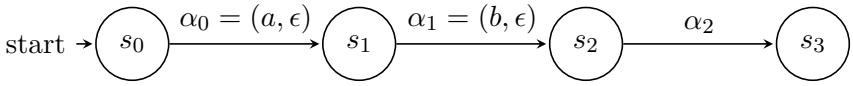
A product Z of a system model and a specification can be used to determine whether that specification would accept or reject the system model — or not know to do either or, yet.

Definition 13. The *verification procedure* is defined as follows:

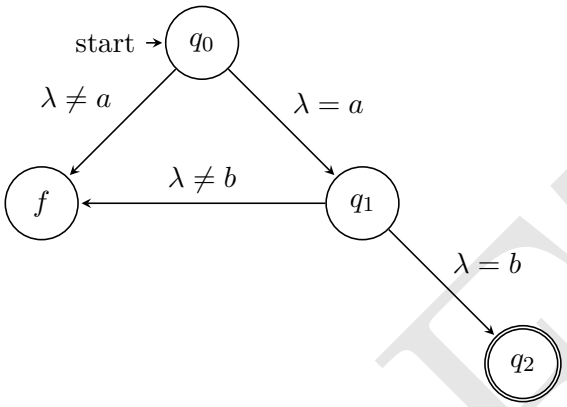
1. Compute the product Z of the specification C and the system model M, as described in Definition 12.

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

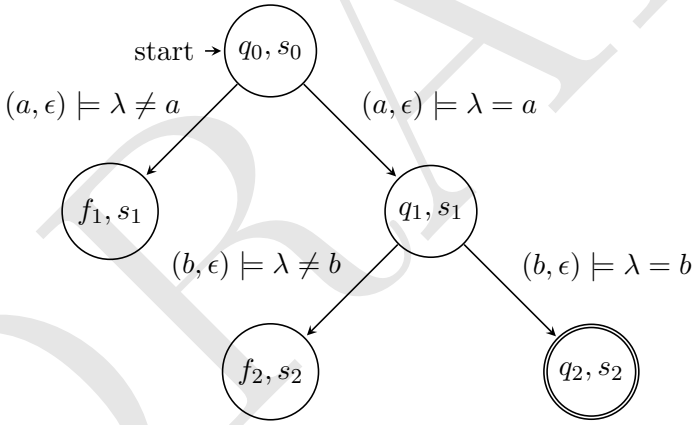
    a) If a fail state is reached, the system model violates the specification.



(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 4.20. Deduction example.

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.

Formal *pythonrv* specifications are verified using this procedure, although in an

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

Figure 4.21. A sketch of the translation procedure of Python expressions to specification expressions. PE is the mapping function, replacing the Python syntax with mathematics syntax.

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.

4.4.4 Examples of Formal Specifications

Here follows two examples of formal *pythonrv* specifications, both in their specification function form, written in Python, shown in Figure 4.22 and Figure 4.24, and as automata, shown in Figure 4.23 and Figure 4.25.

The first example, Figure 4.22 and 4.23, 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 4.6 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 4.24 is more complex, as can be seen by the sprawling automata in Figure 4.25. 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.

```
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(lambda: spec)
  # combine by tail composition
  return a + n
```

Figure 4.22. A formal *pythonrv* specification function, similar to the informal specification function shown in Figure 4.6. The automata for this specification function is shown in Figure 4.23.

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

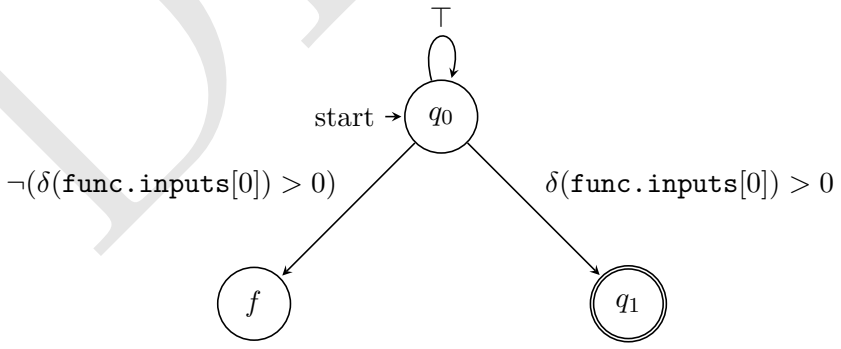


Figure 4.23. The automata for the formal *pythonrv* specification function in Figure 4.22.

```
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-then 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 tail 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 tail composition
  return a + n
```

Figure 4.24. A more complex formal *pythonrv* specification function. The automata for this specification function is shown in Figure 4.25.

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

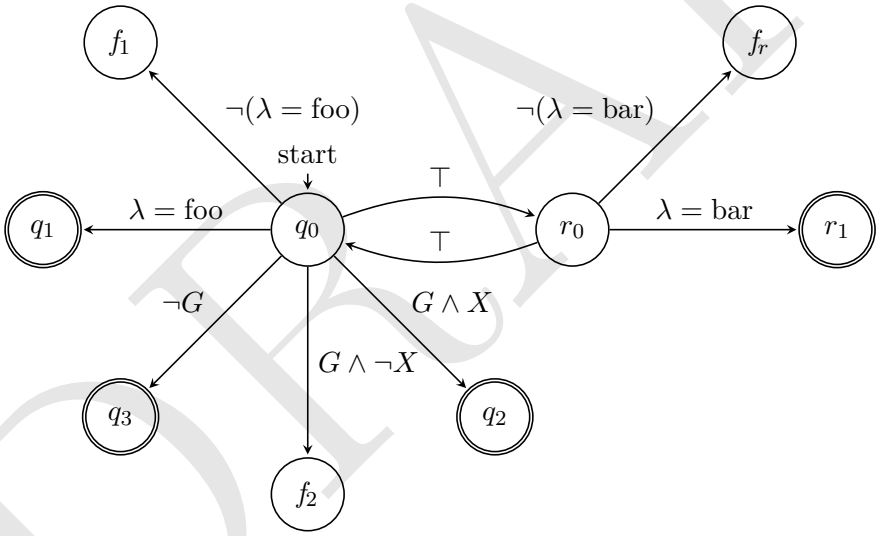


Figure 4.25. A direct translation of the formal *pythonrv* specification function in Figure 4.24 to an automata. q_0 is the root for the `spec` specification, and r_0 is the root of the `spec_bar_called` specification. q_1 and f_1 relates to the assertion in `spec`, and q_2, q_3 and f_2 to the if statement. r_1 and f_r relates to the assertion in `spec_bar_called`.

Chapter 5

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.

5.1 Technical Perspective

5.1.1 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/`

5.1.2 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.

5.1.3 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.

5.2 Potential Value

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 5.1. 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.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.

The same specification can be written in the formal syntax, see Figure 5.2.

```
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)
@rv.spec(when=rv.POST)
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 5.1. An informal *pythonrv* specification verifying that views are only accessible with proper authentication. See Figure 5.2 for an informal version.

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

@rv.monitor(bh=BaseHandler.get_response)
@rv.spec(when=rv.POST)
@formal_spec
def ensure_auth(event):
  auth = make_assert(lambda e:
      e.called_function.inputs[1].user.is_authenticated())
  active = make_assert(lambda e:
      e.called_function.inputs[1].user.is_active)

  return make_if(requires_auth, auth + active)

def requires_auth(event):
  return event.called_funcion.result.status_code == 200 and \
      not event.called_function.inputs[1].path.startswith("/login") and \
      not event.called_function.inputs[1].path.startswith("/appmedia")
```

Figure 5.2. A formal *pythonrv* specification verifying that views are only accessible with proper authentication. See Figure 5.1 for an informal version.

Chapter 6

Conclusions

In this report, and with the proof-of-concept implementation *pythonrv*, we have shown that it is possible to write specifications in the programming language Python and in a manner more similar to unit testing. Our approach utilizes the dynamic nature of Python to do instrumentation. Specifically, we use decorators to mark what functions should be monitored for a specification, and reflection to deduce the properties of the monitorees. We also use the fact that each function has a parent container, so it can be rewritten during runtime with the monitoring code.

However, our approach would work in other programming languages as well if they support hot-swapping of code — replacing objects during runtime, injecting wrapping code around the functions to monitor.

We have also implemented support in *pythonrv* for formal verification. The formal specifications are written in a subset of Python, which is given a formal semantics, see Section 4.4.

A few reservations should be mentioned, however.

The specifications' 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 4.

Giving the specifications a formal foundation, and doing formal verification with them, has been difficult, due to the fact that the chosen programming language, Python, does not have a formal semantics defined. We define the formal semantics for a subset of Python, which makes the math easier, but the resulting syntax less interesting — it requires a lot of boilerplate for even the simplest of specifications.

6.1 Other Approaches

In Chapter 2 we describe approaches that others have taken to runtime verification. The formal specification languages used are mostly LTL or variations thereof, and either written as comments or special attributes in code, or as separate specification files. See e.g. Bauer et al. [11], Bodden [13], Kähkönen et al. [14], Java PathExplorer by Havelund et al. [15], Temporal Rover by Drusinsky [12]; these all use a language

for specifications separate from the target program's programming language.

The informal approaches, like some implementations of Design by Contract, and simple assertions, often use the target program's programming language. The contracts and assertions are often heavily attached to the code they verify, e.g. in [19, 18].

LogScope [24] is one of the few runtime verification approaches that are written in Python. It is very different from *pythonrv* — its specifications are written in a temporal logic based on RuleR, and it builds its system model from log files, working completely offline.

6.2 Future Work

The testing tool called expectations, as described in Section 3.4, could fit quite well with the *pythonrv* style of writing specifications. This could allow for a cleaner and more succinct way to describe temporal properties, perhaps obviating the need to use the `event.next` method in many cases.

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 2.3 and Section 4.3 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.

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.

It would be interesting to investigate how the approach we have used in this report would work in languages beside Python. Both Java and Ruby, for instance, support hot-swapping of code, but an implementation of *pythonrv* in those languages would look very different than the original, due to the underlying workings of those languages.

6.3 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`

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] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] D. Drusinsky, "The temporal rover and the atg rover," in *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, (London, UK, UK), pp. 323–330, Springer-Verlag, 2000.

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

[14] K. Kähkönen, J. Lampinen, K. Heljanko, and I. Niemelä, "The lime interface specification language and runtime monitoring tool," in *Runtime Verification* (S. Bensalem and D. A. Peled, eds.), pp. 93–100, Berlin, Heidelberg: Springer-Verlag, 2009.

[15] K. Havelund and G. Roşu, "An overview of the runtime verification tool java pathexplorer," *Form. Methods Syst. Des.*, vol. 24, pp. 189–215, Mar. 2004.

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

[17] 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.

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

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

[22] M. Matusiak, "Strategies for aspect oriented programming in python," May 2009.

[23] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of aspectj," in *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, (London, UK, UK), pp. 327–353, Springer-Verlag, 2001.

[24] H. Barringer, K. Havelund, D. Rydeheard, and A. Groce, "Rule systems for runtime verification: A short tutorial," in *Runtime Verification* (S. Bensalem and D. A. Peled, eds.), pp. 1–24, Berlin, Heidelberg: Springer-Verlag, 2009.

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

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

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

[28] 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 2.4.1.

mock, mocking Using fake, stand-in objects to isolate units of the program from the rest of the system. See Section 3.3.

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 2.3 for an overview and Chapter 2 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. [9] for more.

specification Something that describes the correct behaviour of something else. See Section 2.4.

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 Chapter 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 3.

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.