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: title? Erland Ranvinge
Supervisor CSC: title Narges Khakpour
Examiner: title Johan Håstad

TRITA xxx yyyy-nn

Abstract

Abstract in English. Write when most of the report is written.

Keywords: Runtime Verification, Unit Testing

Referat

"TODO: Test-inspirerad runtime-verifiering"

Sammanfattning på svenska. Skrivs sist.

Keywords (Sökord? Nyckelord?):

Preface

This is a master thesis / degree project in Computer Science at the Royal Institute of Technology (KTH), Stocholm. The work was done at Valtech Sweden, an IT Consultancy. It was supervised by Erland Ranvinge (Valtech) and Dr. (**todo: check**) Narges Khakpour (CSC KTH).

todo: Thanks to people

Contents

Chapter 1

Introduction

Due to the increasing size and complexity of computer software it has become increasingly difficult, if not impossible, to convince oneself that the software works as desired. This is where verification tools can be used to great effect. Of these tools, testing is the one known by most and in wide spread use. The spread of agile development practices and test-driven development has also popularized the concept of *unit 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 - follow its specification. Formal verification techniques, such as theorem proving, model checking (and its bounded variant), can give such proofs, but they often suffer from complexity problems (incompleteness, undecidability) and practical issues, such as the so-called state explosion problem, and not being fully automated.

A relatively **(rephrase)** new approach in this area is runtime verification, in which the program *execution* is verified against its specification. With the specification written in a suitably formal language, the program can be monitored to check that the specification is followed.

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?

1.2 Motivation

Checking that a program works correctly is of great interest to software developers, and formal verification techniques can often help. As mentioned above, traditional approaches can be impractical with larger programs, and verification by testing is

informal and incomplete. Runtime verification can here be a lightweight addition to the list **(rephrase)** 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 and special tools to support this specialised language's syntax. In contrast, unit testing frameworks often utilise the programming language to great effect, and their use is wide spread.

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 software systems.

1.3 Disposition

Perhaps: Discuss the sectioning of this report.

The rest of this report is structured in as follows. Chapter 2 gives a background to the subject of verifying program correctness. Chapter 3 continues by describing the previous research on runtime verification and the syntax of specification languages. It also gives an overview of the current ideas in unit testing.

What will this report discuss? What problems? Why is this interesting?

What will this report **not** discuss?

Chapter 2

Background

Runtime verification is a new area of research, but the research on verification and formal methods goes back several decades. Research of interest include the early work on formal methods, e.g. by Hoare [1] and Floyd [2], and work on logics suitable for runtime verification, e.g. LTL by Pnueli [3]. The seminal work done by Hoare, Floyd and Pnueli lay the foundation for many interesting approaches used for runtime verification. LTL is one of the common formal languages used for formal specifications in runtime verification.

This chapter gives a short overview of the background to the concepts of this report. It starts with laying out what we mean by proving the correctness of programs in Section 2.1. Section 2.2 describes runtime verification and its place in proving correctness. And finally, Section 2.3 discusses testing.

2.1 Proving 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. There are several approaches, with their advantages and disadvantages.

Theorem proving, as started by Hoare [1], Floyd [2] 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 the post-conditions of the exit point(s) so that they follow the specification, and by linking together the pre-conditions of program points with their preceding program points' post-conditions, we now know that correct indata will yield correct results.

This way of proving correctness often yields the best results. But it is slow, hard to automate, and therefore requires much manual labor. Wading through large programs thus often becomes impractical.

Model checking is the concept of verifying that a *model* of a system 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 a logical formula. It is often simpler than theorem proving, and can be automated.

The model of the system is usually structured as a finite state machine, and verification means visiting all accessible states, checking that they follow the specification. 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. Runtime verification attempts to solve this by dealing directly with the system, creating its model during runtime.

2.2 Runtime Verification

Much in common with model checking. Only current execution. Finite traces. Dynamic environment.

Runtime verification (RV) is a dynamic approach to checking program correctness, in contrast to the more traditional formal static analysis techniques discussed above. These are often very useful, but suffer from severe problems such as the state explosion problem, incompleteness, undecidability etc., when they are used for verification of large-scale systems. Moreover, static analysis usually verifies an abstract model of the program, and cannot guarantee the correctness of the implementation or the dynamic properties of the executing code.

Runtime verification aspires to be a light-weight formal verification technique, see e.g. [8, 9]. It verifies whether some specified properties hold during the execution of a program.

The specification that should be verified is often written in a formal language, often a logic/calculus, such as linear temporal logic [3]. To build a *system model* for verifying the properties of the specification, the target program needs to emit and expose certain events and data. The collected events and data are used to build the system model. Many RV frameworks use *code instrumentation* to generate *monitors* for this end.

There are two types of monitoring: *online* and *offline*. In online monitoring, the analysis and verification is done during the execution, in a synchronous manner with the observed system. In offline monitoring, a log of events is analysed at a later time.

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

Relevant work on runtime verification include [13], in which Bauer et al. use a three-valued boolean logic (true, false and ?) to reflect that a specification can not

only be satisfied (true) or violated (false), but also neither yet, or, in the future it may be either. Bauer et al. also show how they transform specifications into automata (i.e. runtime monitors). Bodden presents in [10] a framework for RV implemented through *aspect-oriented programming* [**?**] in Java, with specifications written as code annotations.

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

2.3 Testing

On the other end of the program-correctness-checking spectrum is *testing*, which is the practical approach of checking that the program, given a certain input, produces the correct output. Testing is not complete, and lacks a formal foundation, so it cannot be used for formal verification. Testing can be a complement to more formal techniques, such as RV (but is in many cases the sole correctness-checking tool).

Unit testing is quite young, perhaps having begun in earnest in the 90s, and it was popularized by the extreme programming (XP) movement **todo: cite someone**. Testing in general is very old.

Kent Beck introduced the style of the modern unit testing framework in his work on a testing framework for Smalltalk [11]. Together with Eric Gamma he later ported it to Java, resulting in JUnit [**?**]. Today, this has lead to frameworks in several programming languages, and they are collectively called xUnit [16].

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

Writing unit tests, often using unit testing *frameworks* such as JUnit [**?**] for Java and unittest [**?**] for Python, is a common practice on many development teams.

Testing is not formal - it doesn't prove anything except that the program works for the provided test cases. Testing is also not complete - for all but the most trivial programs, it is impossible to write tests for all cases.

Testing is often a manual process, taking up a large part of development time (see e.g. [**?**]). Still, there are tools to automatically generate tests.

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 so that it is good. Tests here play the part of specifications for the units of the program.

todo: Find correct places to cite. Original book on TDD?

Chapter 3

Previous Research

As we saw in Section 2.2, runtime verification is the technique of verifying a program's compliance against a specification during runtime. These specifications need to be written somehow, which will be discussed in Section 3.1. Approaches for verification are discussed in Section 3.2. For verification to work, during runtime, the program usually needs to be instrumented in such a way that the verification process can access all pertinent data. This is discussed in Section 3.3

The design of unit test syntax is discussed in Section 3.4. The combination of the two, runtime verification and unit testing, will be the main subject in Chapters 4 and 5.

3.1 Specifications

Specifications come in many forms, from the informal ones like "I want it to be easy to use", to the contractual ones written by companies and clients, to the ones written in formal languages, specifying properties that should verifiably hold about the program. It is this last type of specifications we are interested in here, and which play an important role in runtime verification.

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

3.1.1 Formalisms for Specifications

There are several common formalisms for writing specifications, and many papers that expand, rephrase and illuminate on them. Although they can be quite different, they share a common origin in the work done by Floyd [2], Hoare [1], and others before them. Floyd thought of formulas specifying in/out properties of state-

ments, and chaining these together to form a formal proof for the program. Hoare elaborated on this idea by basing his proofs on a few axioms of the programming language and target computer architecture, and building the proof from there.

Linear Temporal Logic

Linear Temporal Logic (LTL) was first discussed by Pnueli in [3], and has since been popular in many areas dealing with a system model containing a temporal dimension. As Pnueli describes it, it is simpler than other logics, but expressive enough to describe many problems of interest for verification. This has been "confirmed" **(rephrase)** by the diverse use of LTL by many researchers **todo: citations**.

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, 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*, and $\boldsymbol{F}$, *eventually*.

Bauer et al. introduce a three-valued boolean logic LTL_3 in [13], taking 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, can not only be satisfied or violated, but inconclusive. For satisfied or violated specifications, no further verification is required - we already know the outcome. For inconclusive verification, we need to continue verification, as, with future events, the result could be either or.

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

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

Figure 3.1. An example of an LTL formula. This can be read as: Globally, if P holds, then, while P didn't hold, S held at some point.

There is a counterpart to LTL in the real-time setting called Timed Linear Temporal Logic **todo: cite someone**. It introduces clocks to make specifications of real-time properties possible. It is of great interest to runtime verification, but won't be discussed further here. **todo: See [] and [] for more on TLTL**.

Design by Contract

Design by Contract was introduced by Bertrand Meyer in **todo: where**, and has been fully implemented in the Eiffel programming language. A contract is the idea that functions, and methods on objects, promise to fulfill certain post-conditions

(or promises) if the inputs they are given fulfill the pre-conditions (or requirements) 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.

Design by Contract is inspired by Hoare logic, and is essentially Hoare logic written in a certain style.

Assertions

A common construct that is part of many popular programming languges, like C, Java, Python, is the assertion statement. It is a way to assert that some predicate holds at a point in the program. Usually the predicate is an expression of the programming language, and is not supposed to alter the program state.

Assertions are distinct from the normal program flow, and not to be confused **(rephrase)** with exceptions. Assertions check for properties that should always be true, anything else would be a programming error.

3.1.2 Writing Specifications

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

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

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

Rosuenblum [6] uses specially annotated comments. Bodden [10] uses Java annotations, which are written at function and variable definitions, to mark code for verification. **todo: more, and rephrase**. The programming language Eiffel has full language support for Design by Contract, with pre- and post-conditions, invariants, and more. Other approaches **todo: who?** use external specification files. For simple cases it is common to write assertions in the program, checking boolean expressions under runtime **todo: cite jass**.

3.2 Verification against Specifications

Formal specifications are written so that programs can be verified against them - to see whether they follow the specification, or violate parts of it.

There are several ways to verify a program against its specification. A common one, used in [13, 10] among others, is to generate state machines from the specification. These state machines, often called *monitors*, operate with the input language of events emitted by the program.

todo: Monitors. Büchi Automatons.

3.3 Code Instrumentation

For verification to work, the verifier (such as a monitor) needs access to events happening in the program. Such events can be functions called, statements executed, variables assigned, etc., depending on the system model of the specification language **(rephrase)** . 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 "sent" to the verification tools.

3.3.1 Pre-processing the Code

Rosuenblum [6] uses a pre-processor step in the C compilation setup instrument code, where the specifications (called assertions there) are written adjacent to the code under watch.

3.3.2 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.

3.3.3 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 delegates to it.

3.3.4 Aspects

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

Runtime verification is a typical case of a cross-cutting concern. Bodden [10] uses it in his runtime verification implementation.

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

3.4 Unit Testing

We discussed testing and unit testing in general in Section 2.3. How does unit testing work? What is the syntaxes? This section mostly concerns the language and syntax used for writing unit tests.

3.4.1 xUnit

The xUnit style of unit testing [16] 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, and one of the first, implementation, I'll use it for a short demonstration. See Figure 3.2.

In JUnit, and xUnit, you run a *test suite* of *test cases*, which contain tests. The example in Figure 3.2, 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, showing progress to the user. When the tests are finished, any errors are shown to the user.

The example in Figure 3.2 has two tests, and methods to set up and tear down the tests *fixture*. The fixture is the surrounding set of objects (environment) that the object under test requires to work properly. These functions are usually called *setUp* and *tearDown*, respectively.

The order in which tests are executed is arbitrary in JUnit. One possible ordering of the relevant methods in the example is:

1. *setUp*
2. *testSimpleAddition*
3. *tearDown*
4. *setUp*
5. *testThatDoWorkReturnsX*
6. *tearDown*

Test written in this style are traditional unit tests.
todo: Mocking? This is "TDD"-style

```
// 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 3.2. An example of unit testing syntax, written as a test case for JUnit.

3.4.2 Behaviour-driven Development

There is a style of writing tests called behaviour-driven development [**?**]. It originated from test-driven development, and is built on the idea that the tests you write should test the behaviour of the program. The simplest example is that you write your unit tests after the behaviour you desire, perhaps naming your tests according

to "X should do Y". A more radical example is shown in Figure 3.3.

```
+Scenario 1: Account is in credit+
Given the account is in credit
And the card is valid
And the dispenser contains cash
When the customer requests cash
Then ensure the account is debited
And ensure cash is dispensed
And ensure the card is returned
```

Figure 3.3. An example scenario describing a behaviour, as written in BDD. Scenario taken from [**?**].

The test runner parses each scenario, and for each line finds a matching unit of code that does what the line describes. This way of writing tests, or describing behaviours, leads to a outside-in, or top-down, way of writing and thinking about your program.

3.4.3 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/data/-configuration Z. Z might be something simple, which we can easily create in the test. It might also be a network or database connection, or something doing heavy calculation, or just something complex.

One way to work around this is to create fake/mock/dummy, objects. A fake network connection has the same interface as a real network connection, but calling it doesn't actually transmit anything anywhere, and it might return 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.

3.4.4 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* [**?**]. A simple example of expectations is shown in Figure 3.4.

Figure 3.4 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:

- Object (optional) and function.
- Invocation count, how often is the function expected to be called

```
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 3.4. An example of expectations, written using jMock and JUnit. Example taken from [**?**].

- Expected arguments. Can be explicit values, or generic types, or ignored.
- What should happen, e.g. what the function should return.
- When this 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 3.4 shows an example of how to explicitly specify expectations.

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

There are several frameworks for working with expectations, such as jMock[1] for

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

Java, Rhino Mocks[2] and Ludibrio[3] for Python.

[2] http://ayende.com/wiki/Rhino+Mocks.ashx
[3] https://github.com/nsigustavo/ludibrio/

Chapter 4

Method

This chapter describes the proof-of-concept implementation of this report.

4.1 Introduction

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

1. How should the syntax for the specification be defined, so that it looks similar to that for unit tests, but works for runtime verification? Which language could be used? Which unit testing framework to take inspiration from?
2. How should the program be instrumented to monitor the system, to expose the appropriate events and data, and to build the system model?
3. How will this be used to verify the system against the specification? Online or offline verification? E.g. which techniques should be used to verify the monitored system against the specification?
4. How can it be provided with a formal foundation?

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

4.1.1 Definitions

Define some stuff first? Instrumentation? "The function to be monitored"? Specification function? Specification?

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

4.2 Syntax

During the development of this proof-of-concept, the biggest factor in deciding what language to use was how it would assist in instrumentation. Instrumentation is discussed in Section 4.3. Python was chosen as the implementation language.

The canonical framework for doing unit testing in Python is the unittest2 framework that is included in all modern versions. 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 unittest2, and mostly add new miscellaneous features, and better test reporting. The original structure of the unit tests is still prevalent. unittest2 builds on the xUnit style, discussed in Section 3.4.

A few simple example below will introduce the syntax of *pythonrv*. After that the capabilities and limitations of the current implementation will be exposited.

4.2.1 Three Examples

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

The example in Figure 4.1 shows the basics of a *pythonrv* specification. Line 1 imports the rv module from the *pythonrv* package. On line 2 it imports the module(s) containing the function(s) we wish to monitor. Line 5 defines the specification as an ordinary python function called `spec`, taking one argument, `event`. The instrumentation is done line 4 by using the *function decorator*[2] `rv.monitor`. `rv.monitor` declares that the function `fib` in `fibmodule` should be monitored, and, whenever `fib` is called, `spec` should be called as well.

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

The specification function in Figure 4.1 will be called upon every invocation to `fibmodule.fib`.

[2]See Section 4.3 for an explanation of function decorators.

```
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
19   else:
20     # this will never happen
21     assert False, "Neither foo nor bar was called"
```

Figure 4.2. A specification that monitors two functions, `mymodule.foo` and `mymodule.bar`. It asserts that calls to the two functions alternate; that no two calls to `foo` occurs without a call to `bar` in between, and vice versa. The first call has to be to `foo`.

Figure 4.2 shows how a specification can monitor two functions. The specification function will be called whenever either of the monitored functions is called. Which function was called can be determined from the `event` argument, as is done on lines 6 and 9. It is the `called` attribute of a function in the `event.fn` structure allows for this.

The example also shows how the specification can access a history of previous events - events that it has handled in the past. `event.history` is a list of all events that has occurred - that this specification monitors. The last element is the current event, and the next-to-last element is the previous element, which can also be accessed as `event.prev`.

Figure 4.3 shows a more advanced example, in which the `next` function of the `event` argument is used. `next` allows the specification function to add more specification functions (possibly implemented as closures or lambdas) to be executed when the next event occurs.

On line 9 the function `followup` is added to be executed on the next event.

```
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 next event
9       event.next(followup)
10      event.finish()
11    else:
12      # verification has failed
13      # same as assert False
14      event.failure()
15
16  def followup(event):
17    if event.fn.bar.called:
18      event.success()
19    elif event.fn.baz.called:
20      assert event.fn.baz.inputs[0] == True
21      # call this function on next event as well
22      event.next(followup)
23    else:
24      event.failure()
```

Figure 4.3. A more complex example: A specification function that monitors three functions, `foo`, `bar` and `baz`, and makes sure that `foo` is called first, then any number calls to `bar` with the first argument as `True`, and then finally a call to `bar`. After that, any calls are allowed - the specification function won't be verified any longer.

Since `followup` is added in this way - as a "oneshot" specification - it needs to add it self using `next` for verification on multiple events. This is done on line 22.

Figure 4.3 also shows how a specification function can turn its verification off - unsubscribe from future events. `event.finish` and `event.success` are essentially the same, and unsubscribes without further errors. `event.failure` can be thought of as a combination of `event.finish` and `assert False` (which always fails).

4.2.2 Capabilities and Limitations

todo:

Write about the details of the `rv.monitor` and `rv.spec` function decorators, and about the uses of the `event` argument. Also shortly describe how one can configure custom error handlers to take action when verification fails (or do this in

the verification section).

4.3 Instrumentation

The three examples in 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.

The instrumentation is performed 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, possibly modifies it, or uses it in some way (decorates it), and then returns it - or some other function. This is used throughout python to turn functions into static or class methods, for instance. Figure 4.4 shows an example function decorator definition, and Figure 4.5 shows how to use it.

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

Figure 4.4. An example of how to define a function decorator.

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

In python, almost all[3] functions belong to a container of some sorts - a class, an object, or a module. In Figure 4.5 the functions `func_a` and `func_b` belong to the module `test` (the module's name is the same as that of the file containing the code). These containers are essentially dictionaries (*dicts* in python parlance) of key-value pairs, where the keys in this case are function names and the values are objects representing the function code. (There are other types of values in these

```
1   # test.py
2   from function_decorator import decorator
3
4   def func_a():
5     print "a"
6
7   func_a()
8   # output:
9   #   a
10
11  # decorate func_a
12  func_a = decorator(func_a)
13  func_a()
14  # output:
15  #   before
16  #   a
17  #   after
18
19  # decorate func_b - equivalent
20  # to the decoration of func_a
21  @decorator
22  def func_b():
23    print "b"
24
25  func_b()
26  # output:
27  #   before
28  #   b
29  #   after
```

Figure 4.5. An example of how to use the function decorator in Figure 4.4.

containers as well, which we can ignore).

The instrumentation in *pythonrv* works as follows. First, a wrapper function is defined for each function to be monitored. 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 the monitored function. The container of the monitored function is then extracted, and the

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

reference to the monitored function is overwritten with a reference to the wrapper. See Figure **??** for a pseudo-python overview.

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

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

Figure 4.6. An overview of the *pythonrv* instrumentation process, in pseudo-python. This is just for illustrative purposes and not how *pythonrv* actually does the instrumentation.

4.4 Verification

Veriaofsdjfaoisdjfaoisdfjalöskdfjalksfhaklsjdhflaksjhdf löadjsf ölaksdhf öajksdhf lkasdhf kashdf öashf lkajshf ölaksdhf öaklsdf aölksdf öalksdjf aölsdjf aölsdjf alöksdfj aölsdkfj

4.5 Formal Foundation

This is the most difficult part, and I really figured it out yet. What follows is an early attempt at it.

A seemingly insurmountable problem quickly arises when attempting to give a formal foundation to the specifications described in Section 4.5. The specifications are written as ordinary python functions and, as such, are difficult to formalize.

Python as a language is rather informal - one implementation of it, CPython, serves as the reference implementation.

One way to go around this is to define a semantics for a subset of python. Specifications written in this subset will have a formal semantics, and they will have a way to formally prove their correctness.

One formal semantics that can quite easily be ascribed to python code is that of finite automata. The python code is translated into a finite automata. To make this more manageable, translations are shown for some small "types of python specifications", which can be composed to yield more complex specifications. Proofs can thus use structural induction to prove general properties no specifications.

The system model for the automata, the alphabet it works on, is the events of the function(s) that a specification monitors. **todo: Currently the specifications below only allow for one monitored function per specification.**

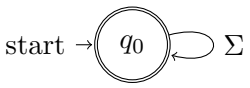
The idea with the transitions in the automata is that either we always follow a transition, and we label it Σ, or we follow a transition only if the event that has occurred satisfies some predicate P. If no transition fits, the specification has been violated.

One peculiarity should be noted: On each new event, a new "instance" of the automaton is spawned, starting from the beginning. All current "executions" of the automaton is also continued.

Next follows the four specification modules I think are required. Their presentation is quite ugly, unfortunately.

```
1  def null_spec():
2    pass
```

(a) The *null specification*, in python function format.

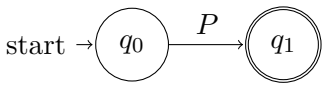


(b) The *null specification*, as an automaton.

Figure 4.7. The *null specification* - the simplest of specifications.

```
1  def assert_spec():
2    assert P
```

(a) The *assert specification*, in python function format.

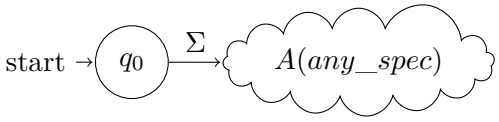


(b) The *assert specification*, as an automaton.

Figure 4.8. The *assert specification* - the simplest specification that actually does something.

```
1        def next_spec():
2          next(any_spec)
```

(a) The *next specification*, in python function format.

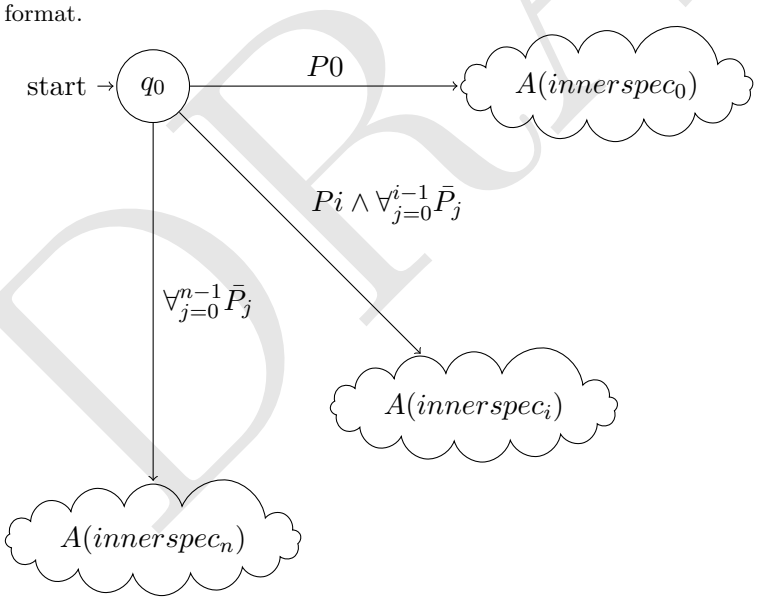


(b) The *next specification*, as an automaton.

Figure 4.9. The *next specification* - the specification that deals with time.

```
1        def if_spec():
2          if P0:
3            innerspec_0() # next or null spec
4          # optional additional conditionals, 1 to n-1
5          else if Pi:
6            innerspec_i() # next or null spec
7          # optional else clause
8          else:
9            innerspec_n() # next or null spec
```

(a) The *if specification*, in python function format.



(b) The *if specification*, as an automaton.

Figure 4.10. The *if specification* - the most complex specification.

Chapter 5

Results

Mm.

Chapter 6

Conclusions

Yay, it worked!

6.1 Discussion

What do we see in the future? How can this be extended, continued?
Results (un)expected? Larger context.
Some speculation? Recommendations?

6.2 Future Work

Some temporary citations: [1], [2], [3], [8], [13], [14], [9], [5], [6], [7], [15], [10], [11], [16], [4]

Bibliography

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

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

[3] 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.

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

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

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

[7] 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).

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

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

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

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

[12] A. Bauer, M. Leucker, and C. Schallhart, "Runtime verification for ltl and tltl," tech. rep., Institut für Informatik, Technische Universität München, December 2007.

[13] 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.

[14] A. Bauer, M. Leucker, and C. Schallhart, "The good, the bad, and the ugly, but how ugly is ugly?," 2008.

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

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