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 design 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

todo: More general background on correctness, testing, unit testing, etc.

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 are among the interesting approaches used for runtime verification. LTL is one of the common formal languages used for formal specifications in RV.

The work on the linear temporal logic (and other logics), on runtime verification in general and its applications, on code instrumentation (e.g. [**?**, 4]), and on unit testing and their frameworks will lay the foundation of this work. Interesting research also include the work by Meyer on the "Design by Contract" methodology [5] and on programming with assertions in general, see e.g. [6, 7].

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.

todo: Formal verification, as started by Hoare [1] and Floydfloyd67 and those around them, is the manual, semi-automated, or (not so often) fully automated process of proving 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 yield according to 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 what? Nice, simpler than formal verification. Can yield

impossibly large state spaces. Bounded model checking.

Requires a model. Can learn model for black box.

todo: How verification tools are used in practice?

2.2 Runtime Verification

todo: Copied from spec, rewrite

The idea: Lightweight formal verification. Execution trace. Speed? Monitoring.

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 of *model checking* (or its bounded form) and *theorem proving*. 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 is 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 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. [**?**]).

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).

Relevant work on runtime verification include [**?**], in which Bauer et al. use a three-valued boolean logic (true, false and ?), and present how to 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

Unit testing is quite young, perhaps having begun in earnest in the 90s, and it is not as much researched as formal methods. 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 *fowlerxunit*.

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.

Not formal - doesn't prove anything except for the specified test cases. Not complete.

Manual. Automatic test-generation?

Test-driven development. Behaviour-driven development.

Chapter 3

Previous Research

todo: Previous work, in RV.

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

Linear Temporal Logic

Linear Temporal Logic (LTL) was first discussed by Pnueli 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 many researchers **todo: citations**.

LTL uses a system model of infinite execution traces, or histories. This fits well into the domain of model checking and theorem proveing. However, due to the necessarily finite nature of runtime verification (no execution of a real-world program can be infinite), Bauer et al. argue in [12, 13] for a slightly modified variant of LTL more suitable for finite execution traces. They name it LTL_3, and the name derives from the fact that they modify the semantics of LTL to not only yield the truth values $\top$ (true) and $\bot$ (false), but also ? (inconclusive).

Given a finite execution trace (a word) u and a LTL_3 formula φ, the truth value of φ with respect to u is denoted by $[u \models \varphi]$, and takes the truth value according to:

$$[u \models \varphi] = \begin{cases} \top & \text{if all (in)finite continuations of } u \text{ would yield } \top \\ \bot & \text{if all (in)finite continuations of } u \text{ would yield } \bot \\ ? & \text{otherwise} \end{cases}$$

todo: Perhaps be more "formal" with symbols

As they write in the paper, this leads to more informative information verifying the specification, and which is particularly well suited for incremental, ongoing, evaluation. In regular LTL, a formula can either be true or false, given an execution trace, but this gives no information of whether it will change in the future.

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**.

EAGLE?

What about EAGLE?

CPL

Hmm, CPL?

Design by Contract

Design by Contract was introduced by Bertrand Meyer in **todo: where**, and has been fully implemented int 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.

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

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.

In Bodden, specifications are written as Java annotations **todo: footnote describing them?** embedded in the target program. Rosuenblum [6] uses specially annotated comments. 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 **(rephrase)** **todo: which**. A common one, used in [13, 10] among others, is to generate monitors from the specification.

Monitors are state machines that operate with the input language of events emitted by the program.

todo: Monitors. Büchi Automatons.

3.3 Code Instrumentation

To 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.

Instrumentation techniques can be divided into two parts: those that require you to manually mark code for "recording", and those that inject the recording code externally. Boddy

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. Bodden [10] uses Java annotations, which are written at function and variable definitions, to mark code for verification. **todo: more, and rephrase**.

For compiled, and byte-compiled **todo: define, another word?**, code, it is possible to rewrite the compiled program to add recording functionality. In dynamic languages, such as Python, Ruby or JavaScript, this can be done dynamically during runtime.

An interesting approach to external injection is to use aspect-oriented programming.

todo: define aspects, join points and point cuts.

3.4 Unit Testing

How do they work? What are their syntaxes? This section mostly concerns the language and syntax used for writing unit tests.

3.4.1 xUnit

JUnit, suites, test cases, set up / tear down. Fixtures? Mocking? This is "TDD"-style

3.4.2 "BDD"-style

describe. it. should. etc.

Chapter 4

Method

What have I done, and why (again)? Test-Inspired Runtime Verification.

4.1 Syntax?

asdf

4.2 Verification, Constructing Monitors

Bauer, Leucker, Schallhart.

4.3 Correctness

It is all awwwesomee!

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.

Appendix A

RDF

And here is a figure

Figure A.1. Several statements describing the same resource.