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 more difficult, if not impossible, to convince oneself that the program works as desired, without the help of verification tools. 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 a program or system is divided into small modules and 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.

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.
todo: How verification tools are used in practice?

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].

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 [8] 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 [9], together with an exposition on the advantages and disadvantages, similarities and differences, with other verification approaches. In [10], Delgado et al. classify and review several different approaches and frameworks to runtime verification.

Unit testing is also 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*.

2.1 Proving Correctness

2.2 Formal Verification

"Best result". Tedious. Often impossible.

2.3 Model Checking

Nice, simpler than formal verification. Can yield impossibly large state spaces. Bounded model checking.

Requires a model. Can learn model for black box.

2.4 Testing

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

Manual. Automatic test-generation?

TDD. BDD.

Chapter 3

Previous Research

todo: Previous work, in RV.

todo: Copied from spec, rewrite

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. [9, 10]. 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).

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.

3.1 Runtime Verification

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

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

3.2 Specifications

High-level, abstract. Easier than the implementation.

3.2.1 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] and [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

Linear Temporal Logic (LTL) and its cousins (such as Timed Linear Temporal Logic, TLTL)

LTL. TLTL. LTL_3.

3.2.2 EAGLE?

What about EAGLE?

3.2.3 CPL

Hmm, CPL?

3.2.4 Design by Contract

Write some about DbC, and how it is written. LTL without the temporal parts.

3.3 Verification against Specifications

Monitors. Büchi Automatons.

3.4 Code Instrumentation

What other stuff is there?

3.4.1 Aspects

AspectJ.

3.5 Unit Testing

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

3.5.1 xUnit

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

3.5.2 "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], [9], [13], [14], [10], [5], [6], [7], [15], [8], [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] E. Bodden, "Efficient and Expressive Runtime Verification for Java," in *Grand Finals of the ACM Student Research Competition 2005*, March 2005.

[9] 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.

[10] 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.

[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.

that we refer to here: A.1