Degree project specification

Adam Renberg
adamre@kth.se

June 23, 2012

School: School of Computer Science and Communication (CSC) at KTH
Company: Valtech, http://www.valtech.se/

Supervisor, Valtech:	Erland Ranvinge	erland.ranvinge@valtech.se
Supervisor, CSC:	Narges Khakpour	nargeskh@kth.se
Examiner:	Johan Håstad	johanh@kth.se
Coordinator, KTH:	Ann Bengtsson	ann@csc.kth.se

Contents

1 Short Background

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. [1, 2]. 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. [4]).

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 [5] for

Java and unittest [6] for Python, is a common practice on many development teams.

2 Problem

2.1 Problem Definition

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?

Suggested title: *Test-inspired runtime verification - using a unit test-like specification syntax for runtime verification.*

2.2 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.

The specification languages used by RV 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 syntax. In contrast, unit testing frameworks often utilise the programming language to great effect, and their use is wide spread.

If RV 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.

3 Literature

3.1 Previous Research

I have already studied several papers, mostly in runtime verification, verification tools, logics, etc. A list of these can be found at the project's website: `http://tgwizard.github.com/thesis/`. Studying all those papers is beyond the scope of this document. Therefore, I will restrict myself to briefly refer to some of the most relevant work here.

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 [7] and Floyd [8], 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. [15, 16]), 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 [12] and on programming with assertions in general, see e.g. [13, 14].

Relevant work on runtime verification include [9], 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* [15] in Java, with specifications written as code annotations.

Leucker et al. present a definition of RV in [1], together with an exposition on the advantages and disadvantages, similarities and differences, with other verification approaches. In [2], 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 [5]. Today, this has lead to frameworks in several programming languages, and they are collectively

called *xUnit*.

3.2 Examination

The literature study will be examined as a part of the report, as the report will contain both a general description of RV and unit testing, and more detailed sections on research related to the problem.

4 Method

The work will consist of two parts:

4.1 Approach

I will do a background and state-of-the-art inventory of RV and unit testing. The focus will lie on the syntax used for writing the specifications, how the properties are verified against the system model, and how code instrumentation is done. I will also focus on the syntax and structure of unit tests, which will have a large bearing on this project.

I will also answer the following questions:

1. How should the syntax for the specification be defined, so that it looks similar to that for unit tests, but works for RV? Which language could be used? Which unit testing framework to take inspiration from?
2. How can it be provided with a formal foundation (e.g. by translating it into a formal logic)?
3. How should the code be instrumented to monitor the system and build the system model?
4. How will this be used to (online) verify the system against the specification? (e.g. which techniques should be used to verify the monitored system against the specification?)

Steps one and three are of most interest to this project, as well as step 2 for the formal foundation, but all four steps are important and required parts of runtime verification.

4.2 Evaluation

In this part I will implement an RV framework prototype, based on the previous investigation.

If possible, I will also attempt to use the resulting framework to enable runtime verification on a project at Valtech.

5 Resources

If I attempt to try the prototype framework on a project at Valtech, I need to get access to such a project. One suggested possible project is the Valtech Intranet.

6 Working Procedure

6.1 Overview

I will work on the degree project 50% and 50% on projects at Valtech, in periods of two weeks degree project, two weeks work. This fits well into the iteration-planning at Valtech. During the summer I will almost exclusively work on the degree project, and also use some of my vacation hours to work on it.

Other required work related to the degree project, such as doing the opposition of another student's work, will take time from the work above and require flexibility in planning. The periods outlined below will of course overlap somewhat, especially in the investigation and implementation parts.

I will keep a diary, `http://tgwizard.github.com/thesis/`, in which I will write on my progress.

6.2 Approximate Time Plan

The literature survey includes: runtime verification approaches, definition of semantics, test approaches/frameworks, formal verification approaches.

Task	**May**	**Jun**	**Jul**	**Aug**	**Sep**	**Oct**	**Total**
Preparing specification	1w	0.5w					1.5w
Literature survey	2w	2w	1.5w				5.5w
Write report (background)			1w				1w
Language syntax design			1w	0.5w			1.5w
Provide formal semantics			0.5w	1w			1.5w
Develop verification approach				1w			1w
Propose approach for code instrumentation				1w			1w
Implementation of RV framework					2w		2w
Evaluating approach					1.5w		1.5w
Write report (all)					1.5w	2w	3.5w
Total	3w	2.5w	4w	3.5w	5w	2w	20w

Table 1: Time plan week-by-week

The evaluation consists of applying the prototype framework on a (possibly real-life) framework.

I will take the weeks v24 and v25 off, as well as about 1 week in august (not planned). **The weeks I will be working at Valtech haven't been written into the time table.**

Total: 20 weeks. End date: **12 October 2012** (very optimistic).

7 Licensing

I will license any resulting code under some open source license, and any documentation, including the report, under Creative Commons Attribution 3.0 Unported License [17] (or something similar).

References

[1] Martin Leucker, Christian Schallhart. A brief account of runtime verification. In *The Journal of Logic and Algebraic Programming 78*. 2009, pages 293-303.

[2] Nelly Delgado, Ann Q. Gates, Steve Roach. A Taxonomy and Catalog of Runtime Software-Fault Monitoring Tools. In *IEEE Transactions on Software Engineering*. IEEE, 2004.

[3] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77)*. IEEE, 1977

[4] Markus C. Huebscher, Julie A. McCann. A survey of Autonomic Computing degrees, models and applications. In *ACM Computing Surveys*, Volume 40 Issue 3. ACM, 2008.

[5] Kent Beck, Eric Gamma, David Saff. JUnit. `http://junit.sourceforge.net/`, Retrieved on June 2012.

[6] Python Software Foundation. unittest. `http://docs.python.org/library/unittest.html`, Retrieved on June 2012.

[7] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. In *Communications of the ACM*, Volume 12 Issue 10. ACM, 1969

[8] Robert W. Floyd. Assigning meaning to programs. In *Proceedings of Symposium on Applied Mathematics*, Vol. 19. A.M.S., 1967

[9] Andreas Bauer, Martin Leucker, Christian Schallhart. Monitoring of real-time properties. In *Proceedings of the 26th Conference on Foundations of Software Technology and Theoretical Computer Science* (FSTTCS), volume 4337 of LNCS. Springer, 2006.

[10] Eric Bodden. Efficient and Expressive Runtime Verification for Java. In Grand Finals of the ACM Student Research Competition 2005. ACM, 2005.

[11] Kent Beck. Simple Smalltalk Testing: With Patterns `http://www.xprogramming.com/testfram.htm`, Retrieved on June 2012.

[12] Bertrand Meyer. Applying "Design by Contract". In *Computer (IEEE)*, 25, 10. IEEE, 1992.

[13] Davis S. Rosenblum. A Practical Approach to Programming With Assertions. in *IEEE Transactions on Software Engineering*, Vol. 21, No. 1. IEEE, 1995.

[14] Detlef Bartetzko, Clemens Fischer, Michael Möller, Heike Wehrheim. Jass - Java with Assertions. In *Electronic Notes in Theoretical Computer Science*, Volume 55, Issue 2. Elsevier 2001.

[15] AspectJ. `http://www.eclipse.org/aspectj/`, Retrieved on June 2012

[16] Martin Matusiak. Strategies for aspect oriented programming in Python. 2009

[17] Creative Commons. Creative Commons Attribution 3.0 Unported License. `http://creativecommons.org/licenses/by/3.0/`