Degree project specification

Adam Renberg
adamre@kth.se

June 11, 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 static analysis tools of *model checking*, *bounded reachability* and *theorem proving*. These are often very useful, but suffers from severe problems such as the state explosion problem, incompleteness, undecidability and the overall difficulty of using them on large programs. Moreover, static analysis only verifies the 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 that specified properties hold during the execution of a program. It operates on a *trace* of the *current execution*, either while it is running (online), or at some other time and/or place (offline), and verifies the execution against a *specification*.

The specification that should be verified is written in a formal language, often a logic/calculus, such as linear temporal logic [3]. To generate a *system model* for the logic (or an automaton thereof) to operate on, the target program needs to emit and expose certain events and data. Many RV frameworks use *code instrumentation* to generate *monitors* for this end.

When a violation against 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 formal techniques, such as RV (but is in many places the sole correctness-checking tool).

Unit testing is the concept of writing small tests, or test suites, for the units in a program, be it 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 resembles the syntax of the target program's programming language, and 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. Runtime verification can here be a lightweight addition to the list.

The specification languages used by RV approaches are often based on formal logic and not written in the target program's programming language. This means that writing the specifications require 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 utilize 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 popularize the use of runtime verification.

3 Literature

3.1 Previous Research

Runtime verification is a somewhat new area of research, but the research on verification and formal methods goes back several centuries.

Unit testing is also quite young, perhaps having started in the 90s, and it isn't as much researched as formal methods. Testing in general is very old.

The work on the linear temporal logic (and other logics), on runtime verification in general and its applications, on code instrumentation (e.g. [7, 8]), and on unit testing and their frameworks will lay the foundation of this work.

I've 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`.

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/tests, and how they are executed and the target program instrumented.

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 this be related to a formal logic, to ensure the correctness of the specification? What formal logic should be used?
3. How should the code be instrumented to monitor the system and build the system model? How should the specification be transformed so it can be executed?
4. How will this be used to (online) verify the system against the specification?

I will focus on item 1 and 3.

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

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.

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

6.1 Approximate Time Plan

Start	Weeks		Work
May	*1w in May/June*	**1w**	Writing, and doing the research for, this specification.
May	*2w in May*, v23, v26, v27	**5w**	Doing background & research.
9/7	v28	**1w**	Writing the background part of the report.
16/7	v29, v30, v31, v32, v33	**5w**	Investigating and evaluating "what to do".
13/8	v34	**1w**	Writing about the investigation.
20/8	v35, v36, v37, v38	**4w**	Implementing and evaluating the RV framework. Possibly testing and analysing the RV framework on a project.
24/9	v39, v40, v41	**3w**	Writing and finishing the report.

vXX means week-of-the-year XX, calculated the Swedish way. I.e., week 28 starts on the 9th of July.

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 [9] (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/`, June 2012.

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

[7] AspectJ. `http://www.eclipse.org/aspectj/`, June 2012

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

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