Requirements-Driven Log Analysis Extended Abstract

Klaus Havelund*

Jet Propulsion Laboratory
California Institute of Technology
California, USA

1 Background

Imagine that you are tasked to help a project improve their testing effort. In a realistic scenario it will quickly become clear, that having an impact is difficult. First of all, it will likely be a challenge to suggest an alternative approach which is significantly more automated and/or more effective than current practice. The reality is that an average software system has a complex input/output behavior. An automated testing approach will have to auto-generate test cases, each being a pair (i, o) consisting of a test input i and an oracle o. The test input i has to be somewhat meaningful, and the oracle o can be very complicated to compute. Second, even in the case where some testing technology has been developed that might improve current practice, it is then likely difficult to completely change the current behavior of the testing team unless the technique is obviously superior and does everything already done by existing technology.

So is there an easier way to incorporate formal methods-based approaches than the full fledged test revolution? Fortunately the answer is affirmative. A relatively simple approach is to benefit from possibly already existing logging infrastructure, which after all is part of most systems put in production. A log is a sequence of events, generated by special log recording statements, most often manually inserted in the code by the programmers. An event can be considered as a data record: a mapping from field names to values. We can analyze such a log using formal methods, for example checking it against a formal specification. This separates running the system from analyzing its behavior. It is not meant as an alternative to testing since it does not address the important input generation problem. However, it offers a solution which testing teams might accept since it has low impact on the existing process. A single person might be assigned to perform such log analysis, compared to the entire testing team changing behavior.

Note that although logging often is manually programmed, it can be performed using automated code instrumentation, using for example aspect-oriented programming. The point here, however, is that manual logging is often already

* Part of the work described in this publication was carried out at Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

done by programmers, and we can try to benefit from this. Analyzing program executions using formal methods is also referred to as *runtime verification* (RV).

2 LogScope and TraceContract

At Jet Propulsion Laboratory (JPL) some positive, although still limited, success with this approach has been gained. Two different systems for analyzing logs against formal specifications have been developed and applied: LogScope and TraceContract. These systems in turn are inspired by previous work, most specifically Ruler [6].

LogScope [3, 8] checks logs against specifications written in a so-called *external DSL* - a stand-alone small Domain Specific Language with its own parser. It is based on a parameterized automaton formalism (conceptually a subset of Ruler) with a second layer of temporal logic which is translated to the automaton level. The temporal logic formalism is very simple and intuitive to use. LogScope was used for a short period by the testing team for the MSL (Mars Science Laboratory) rover, also named Curiosity [12], which landed on Mars on August 5, 2012. LogScope was usable due to its simplicity and ease of adoption. The testing team was, however, shut down during a period due to an otherwise unrelated 2 year delay in the mission, and LogScope was not used when a new team was built later. The application of LogScope on MSL was reported in [3, 8].

TraceContract [4, 5] is a so-called *internal DSL* (an extension of an existing programming language), an API in the Scala programming language, offering an interesting combination of data parameterized state machines and temporal logic. It is currently being tried out by the testing team on the SMAP project [13] at JPL (a future earth orbiting satellite measuring soil moisture), and by the LADEE project [10] at NASA Ames Research Center (a future moon orbiting satellite measuring dust in the lunar atmosphere). The attraction of TraceContract is the expressiveness of the logic, in large part caused by it being an extension of a high-level modern programming language. As such TraceContract represents the use of an advanced programming language for modeling, an interesting point in itself, as also pointed out in [9]. Furthermore, TraceContract has a very small implementation and is exceptionally easy to modify compared to LogScope. We shall discuss the two applications of TraceContract to SMAP and LADEE and compare to the previous application of LogScope to MSL.

3 Requirements Engineering and Logging

We shall furthermore discuss the possibility of relating requirements engineering to logging, and thereby log analysis. A natural thought is to formulate requirements as statements, even informal, involving concrete events (data records), and then enforce programmers to log such events. Requirements can consequently be

converted into monitors and tested on the running system. As an example, consider the informal requirement:

> **Requirement** *If a resource is granted to a task, the resource cannot be granted to some other task without being canceled first by the first task.*

We could formulate this requirement in terms of two formalized event types:

- *Grant(t,r)* : task t is granted the resource r.
- *Cancel(t,r)* : task t cancels (hands back) resource r.

The now semi-formal requirement becomes:

> **Requirement** *If a resource is granted to a task with Grant(t,r), the resource cannot be granted to some other task with Grant(_,r) without being canceled first with Cancel(t,r) by the first task.*

Of course, a proper formalization will be more desirable, but even this informal statement in English over formal events can be useful for subsequent testing purposes. A monitor can for example later be programmed in a system such as TraceContract:

```
class GrantToOne extends Monitor[Event] {
  always {
    case Grant(t, r) =>
      watch {
        case Grant(_, `r`) => error
        case Cancel(`t`, `r`) => ok
      }
  }
}
```

A specific monitor, such as GrantToOne above, sub-classes the Monitor class, parameterized with the type of events. The Monitor class in turn offers a collection of methods for writing properties, such as **always** and **watch**, taking partial functions as argument, specified using pattern matching with Scala's **case** statements. This monitor illustrates the mixture of Scala and added DSL constructs.

4 Future Work

Beyond expressiveness and convenience of a logic, efficiency of monitoring is of main importance. The key problem in evaluating a set of monitors given an incoming event is to perform *efficient* matching of the event (and possibly other facts depending on the logic) against conditions in monitors. This becomes particularly challenging when events carry data parameters, as also heavily studied

in state of the art systems [1, 11]. We are investigating the combination of expressiveness and efficiency, with focus on expressiveness, as documented in [2]. The field of Artificial Intelligence (AI) has itself studied a problem very similar to the runtime verification problem, namely *rule-based production systems*, used for example to represent knowledge systems. We are specifically studying the RETE algorithm [7] for its relevance for the RV problem. This includes implementing it in the SCALA programming language, and visualizing its operation on data structures.

References

1. C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittamplan, and J. Tibble. Adding trace matching with free variables to AspectJ. In *OOPSLA'05*. ACM Press, 2005.
2. H. Barringer, Y. Falcone, K. Havelund, G. Reger, and D. Rydeheard. Quantified Event Automata - towards expressive and efficient runtime monitors. In *18th International Symposium on Formal Methods (FM'12), Paris, France, August 27-31, 2012. Proceedings*, volume 7436 of *LNCS*. Springer, 2012.
3. H. Barringer, A. Groce, K. Havelund, and M. Smith. Formal analysis of log files. *Journal of Aerospace Computing, Information, and Communication*, 7(11):365–390, 2010.
4. H. Barringer and K. Havelund. TraceContract: A Scala DSL for trace analysis. In *17th International Symposium on Formal Methods (FM'11), Limerick, Ireland, June 20-24, 2011. Proceedings*, volume 6664 of *LNCS*, pages 57–72. Springer, 2011.
5. H. Barringer, K. Havelund, E. Kurklu, and R. Morris. Checking flight rules with TraceContract: Application of a Scala DSL for trace analysis. In *Scala Days 2011, Stanford University, California*, 2011.
6. H. Barringer, D. E. Rydeheard, and K. Havelund. Rule systems for run-time monitoring: from Eagle to RuleR. *J. Log. Comput.*, 20(3):675–706, 2010.
7. C. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.
8. A. Groce, K. Havelund, and M. H. Smith. From scripts to specifications: the evolution of a flight software testing effort. In *32nd Int. Conference on Software Engineering (ICSE'10), Cape Town, South Africa*, ACM SIG, pages 129–138, 2010.
9. K. Havelund. Closing the gap between specification and programming: VDM^{++} and Scala. In M. Korovina and A. Voronkov, editors, *HOWARD-60: Higher-Order Workshop on Automated Runtime Verification and Debugging*, volume 1 of *EasyChair Proceedings*, December 2011. Manchester, UK.
10. LADEE: Lunar Atmosphere Dust Environment Explorer. http://www.nasa.gov/mission_pages/LADEE/main.
11. P. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu. An overview of the MOP runtime verification framework. *Software Tools for Technology Transfer (STTT)*, 14(3):249–289, 2012.
12. MSL: Mars Science Laboratory. http://mars.jpl.nasa.gov/msl.
13. SMAP: Soil Moisture Active Passive. http://smap.jpl.nasa.gov.