An Entry Point for Formal Methods: Specification and Analysis of Event Logs

Howard Barringer
School of Computer Science
University of Manchester
Manchester, UK
howard.barringer@manchester.ac.uk

Alex Groce
School of Electrical Engineering and Computer Science
Oregon State University
Corvallis, USA
alex@eecs.oregonstate.edu

Klaus Havelund Margaret Smith
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, USA
klaus.havelund@jpl.nasa.gov margaret.h.smith@jpl.nasa.gov

Formal specification languages have long languished, due to the grave scalability problems faced by complete verification methods. Runtime verification promises to use formal specifications to automate part of the more scalable art of testing, but has not been widely applied to real systems, and often falters due to the cost and complexity of instrumentation for online monitoring. In this paper we discuss work in progress to apply an event-based specification system to the logging mechanism of the Mars Science Laboratory mission at JPL. By focusing on log analysis, we exploit the "instrumentation" already implemented and required for communicating with the spacecraft. We argue that this work both shows a practical method for using formal specifications in testing and opens interesting research avenues, including a challenging specification learning problem.

1 Introduction

NASA's Mars Science Laboratory Mission (MSL), now scheduled to launch in 2011 [1], relies on a number of different mechanisms used to command the spacecraft from Earth and to understand the behavior of the spacecraft and rover. The primary elements of this communication system are commands sent from the ground, visible events emitted by flight software (essentially formalized `printfs` in the code) [7], snapshots of the spacecraft state, and data products — files downlinked to earth (e.g., images of Mars or science instrument data). All of these elements may be thought of as spacecraft *events*, with a canonical timestamp. Testing the flight software (beyond the unit testing done by module developers) usually relies on observing these indicators of spacecraft behavior. As expected, test engineers cannot "eyeball" the hundreds of thousands of events generated in even short tests of such a complex system. Previously, test automation has relied on ad hoc methods — hand-coded Python [9] scripts using a framework to query the ground communications system for various kinds of events, as a test proceeds. This was the state-of-the-art at the time members of our group, the Laboratory for Reliable Software, joined the MSL team, as developers and test infrastructure experts. Lengthy collaboration with test engineers convinced us that something better was required: the test script approach required large amounts of effort, much of it duplicated, and often failed due to changes in event timing and limitations of the query framework. Moreover, the scripts, combining test input activity, telemetry gathering, and test evaluation, proved difficult to read — for other test engineers and for developers and systems engineers trying to extract and understand (and perhaps fix) a specification. Runtime verification using formal specifications offered a solution, and the MSL ground communications system suggested that we exploit an under-used approach to runtime verification: offline examination of event logs already produced by system execution. In the remainder of this paper we report on two aspects of this project — in Section 2 we discuss the general idea of event sequences as requirements, and our specification methodology; Section 3 gives a brief introduction to our application of these general ideas to MSL and our new specification language.

2 Event Sequences as Requirements

Systems verification consists of proving that an artifact (hardware and/or software) satisfies a specification. In mathematical terms we have a model $M \in ML$ (for example the complete system) in some model language ML, and a specification $S \in SL$ in some specification language SL, and we want to show that the pair (M,S) is member of the satisfaction relation $\models \subseteq ML \times SL$, also typically written: $M \models S$. The general problem of demonstrating correctness of a combined hardware/software system is very hard, as is well-known. Advanced techniques such as theorem proving or model checking tend not to scale for practical purposes. Extracting abstract models of the system, and proving these correct, has been shown to sometimes be useful, but also faces scalability problems when dealing with real systems or complex properties. The problem is inherently difficult because the models are complex – because the behavior of systems is complex. The problems of full verification have long limited the adoption of formal specification.

In *runtime verification* a specification is used to analyze a single execution of a system, not the complete behavior. In this case a model is a single execution trace σ, and the verification problem consists of demonstrating that $\sigma \models S$, a much simpler problem with scalable solutions. The original model M (the full system) can be considered as denoting the set of all possible runs σ, of which we now only verify a subset. This approach of course is less ambitious, but seems to be a practical and useful deployment of formal specification. Though these observations are well known, they are less often taken into practice. It is still rare to observe the application of formal specification — even for runtime verification. There may be several reasons for this, one of which is the problem of program instrumentation. A large body of research considers the problem: how to we produce traces to verify? There is, however, a much simpler approach, namely to use logging information that is already generated by almost any computer system when it is tested. Our *first recommendation* is therefore that runtime verification should be used to formally check logged data. Our *second recommendation* is that logging and requirements engineering should be connected in the sense that requirements should be testable through runtime verification of logs. The common element is the *event*: requirements should be expressed as predicates on sequences of events, and logging should produce such sequences of events, thereby making the requirements testable. This means that logs preferably should consist of events with a formal template, connected to the original requirements. Note, however, that it is possible to extract formalized events from chaotic logging information produced by the usual ad hoc methods — using, e.g., regular expressions.

The obvious scientific and methodological question is: what kind of information should a log contain, in order to help verify requirements? We shall adopt the simple view that a log is a sequence of events, where an event is a mapping from names to values of various types (integers, strings, etc.):

$$\begin{aligned} Log &= Event^* \\ Event &= Name \to Value \end{aligned}$$

This definition is quite general. An event can carry information about various aspects of the observed system. Events can be classified into various kinds by letting a designated name, i.e. *kind*, map to the kind. In the context of MSL, five forms of events are used, see Figure 1. We claim that these five event kinds are generally applicable to any system being monitored. The five forms of events are: (1) commands (input) issued to the monitored system, (2) products (output) delivered by the system, (3) periodic samplings of the state of the system, such as the value of continuous-valued sensors (like position coordinates), (4) changes to the state of the system, those changes that are observable, and (5) transitions performed by the system (also referred to as EVent Reports - EVRs), for example when `printf` statements would normally be used to record an important event. The two forms of observation of the state (3, 4) could potentially be regarded as one kind of observation: that of the state of the system at any point in time. The state observations (3, 4) and the transitions (5) are internal events, while the commands (1) and products (2) are external events.

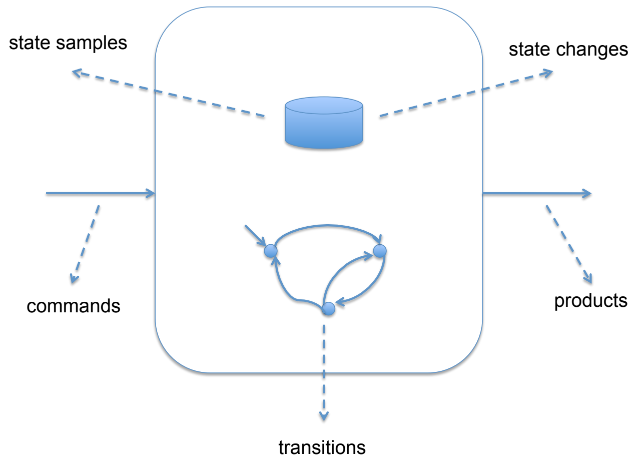

Figure 1: observable events of a system

We have developed a specification language, and corresponding monitoring system, to be presented in the next section, which have been applied by testing engineers within the MSL project. The specification language consists of a mixture of automata and temporal logic with elements of regular expressions. The logic can specifically refer to the data in events. This mixture seems to be attractive for the engineers. In the longer term, we argue that such a specification language and monitoring system should be used in combination with a systematic logging discipline, such that *requirements are formulated in terms of events*, and, minimally, *these events* should be produced as part of logging. These observations may not appear to be ground-breaking, and to some extent have the flavor of *"yes of course"* — but as it turns out, considering current practices in software projects, they may be rather ground-breaking. A successful formal verification story always relies on finding the proper mixture of formality and common practice, as well as the right specification language for the task. We believe that the work described in this short paper has shown the potential for being such a success story.

```
pattern CommandSuccess:
  COMMAND{Type : "FlightSoftwareCommand", Stem : x, Number : y} =>
     {
        EVR{Dispatch : x, Number : y},
        [
           EVR{Success : x, Number : y},
           not EVR{Success : x, Number : y}
        ],
        not EVR{DispatchFailure : x, Number : y},
        not EVR{Failure : x, Number : y}
     }
```

Figure 2: A generic specification for flight software commands.

3 Framework

MSL's ground software stores all events in a SQL database, which we interpret as a chronologically ordered sequence of events — a log. Our Python framework, called LOGSCOPE, allows us to check logs for conformance to a specification and to "learn" patterns from logs. The architecture of LOGSCOPE divides functionality into a LOGMAKER tool, specific to MSL, and a core LOGSCOPE module for checking logs and learning specifications, which may be applied to any ordered event sequence.

3.1 LogMaker

LOGMAKER communicates with MSL's SQL-based ground software to generate a list of events, where each *event* is a record mapping field names to their values. A special field indicates the type of the event: command, transition (EVR), state sampling, state change, or data product. Note how the MSL events map to the generalized idea of a monitored system shown in Figure 1. The log extractor sorts events according to spacecraft event times, since the order in which events are received by ground communications software does not correspond to the order in which events are generated on-board (due to varying communication priorities). Further analysis annotates the log with meta-events for ease of use in specification, and uses spacecraft telemetry to assign a spacecraft time to ground events. We hope to extend and exploit previous work on monitoring distributed systems with multiple clocks [10] to influence flight software's use of telemetry to ensure that effective event ordering is always possible.

3.2 Monitoring

The monitoring system of LOGSCOPE takes two arguments: (1) a log generated by *logmaker*, and (2) a specification. Our specification language supplies an expressive rule-based language, which includes support for state machines, and a higher-level (but less expressive) *pattern language*, which is translated into the more expressive rule-based language before monitoring.

Specifications in the pattern language are easy for test engineers and software developers to read and write. Figure 2 illustrates a pattern. The `CommandSuccess` pattern requires that following every command event (meaning a command is issued to the flight software), where the `Type` field has the value `"FlightSoftwareCommand"`, the `Stem` field (the name of the command) has a value `x` (`x` will be *bound* to that value), and the `Number` field has a value `y` (also a binding variable), we must see (`=>`) — in any order, as indicated by set brackets `{...}` — (1) a dispatch of command `x` with the number `y`; (2) a success of `x/y`, and *after that* no more successes — the square brackets `[...]` indicate an ordering of the event constraints. Furthermore, (3) we do not want to see any dispatch failures for the command; and finally (4) we do not want to see any failures for the command.

Interesting features of the language include its mixture (and nesting) of ordered and unordered event sequences of event constraints, including negations, and its support for testing and capturing data values embedded in events. The pattern language is translated into our rule-based language derived from the RULER specification language [4, 5, 3]. A subset of this language defines state machines with parameterized events and states, where a transition may enter many target states — essentially alternating automata with data. The language is also inspired by earlier state-machine oriented specification/monitoring languages, such as RCAT [11] and RMOR [8].

In addition to exact requirements on field values, our language supports user-defined predicates (written in Python) that may take field values and bound variables as arguments, providing very high expressive power. Specifications are visualized with Graphviz [6], and extensive error trace reporting with references to the log files ensures easy interpretation of detected specification violations.

3.3 Learning

LOGSCOPE was well-received by test engineers, and was integrated into MSL flight software testing for two modules shortly after its release. One important result of early use was to alert us to the burden of writing patterns more specific than the kind of generic rule shown above. In order to ease this burden we introduced a facility for *learning* specifications from runs. Consider a test engineer or developer who runs a flight software test one or more times. If these runs have been "*good*" runs he/she can "endorse" (perhaps after making manual modifications) the specification, and it can then be used to monitor subsequent executions. Learning requires a notion of event equality, and users can define which fields should be compared for testing event equality (e.g., exact timing is usually expected to change with new releases and perhaps even new test executions). We have implemented and applied a *concrete learner* which learns the set of all execution sequences seen so far (essentially a "diff" tool for logs). We also expect to learn mappings from commands to events expected in all execution contexts — a pattern based approach, like that of Perracotta [12]. More ambitiously, we hope to incorporate classic automata-learning results [2] in order to generalize specifications.

4 Conclusions and Future Work

The MSL ground control and observation software demonstrates an important concept: many critical systems already implement very powerful logging systems that can be used as a basis for automated evaluation of log files against requirements. Such log files can be analyzed with scripts (programs) written using a scripting (programming) language. However, there seems to be advantages to using a formal specification language, as demonstrated with this work. A systematic study could be needed, investigating to what extent a domain specific language really is required to achieve the added benefit, or whether a well designed Python API (or API in any other programming language) would yield the same benefits.

Acknowledgements

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

Thanks are due to many members of the Mars Science Laboratory Flight Software team, including Chris Delp, Gerard Holzmann, Rajeev Joshi, Cin-Young Lee, Alex Moncada, Cindy Oda, Glenn Reeves, Margaret Smith, Lisa Tatge, Hui Ying Wen, Jesse Wright, and Hyejung Yun.

References

[1] `http://mars.jpl.nasa.gov/msl`.

[2] Dana Angluin (1987): *Learning Regular Sets from Queries and Counterexamples*. *Inf. Comput.* 75(2), p. 87106.

[3] Howard Barringer, Klaus Havelund, David Rydeheard & Alex Groce (2009): *Rule Systems for Runtime Verification: A Short Tutorial*. In: S. Bensalem & D. Peled, editors: *Proc. of the 9th International Workshop on Runtime Verification (RV'09)*, *LNCS* 5779. Springer, pp. 1–24.

[4] Howard Barringer, David Rydeheard & Klaus Havelund (2007): *Rule Systems for Run-Time Monitoring: from Eagle to RuleR*. In: *Proc. of the 7th International Workshop on Runtime Verification (RV'07)*, *LNCS* 4839. Springer, Vancouver, Canada.

[5] Howard Barringer, David Rydeheard & Klaus Havelund (2009): *Rule Systems for Run-Time Monitoring: from Eagle to RuleR*. *Journal of Logic and Computation*. Advance Access published on November 21, 2008. doi:10.1093/logcom/exn076.

[6] GraphViz: `http://www.graphviz.org`.

[7] Alex Groce & Rajeev Joshi (2006): *Exploiting Traces in Program Analysis*. In: *Tools and Algorithms for the Construction and Analysis of Systems*. pp. 379–393.

[8] Klaus Havelund (2008): *Runtime Verification of C Programs*. In: *Proc. of the 1st TestCom/FATES conference*, *LNCS* 5047. Springer, Tokyo, Japan.

[9] Python: `http://www.python.org`.

[10] Koushik Sen, Abhay Vardhan, Gul Agha & Grigore Rosu (2006): *Decentralized Runtime Analysis of Multithreaded Applications*. In: *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

[11] Margaret Smith & Klaus Havelund (2008): *Requirements Capture with RCAT*. In: *16th IEEE International Requirements Engineering Conference (RE'08)*, IEEE Computer Society. Barcelona, Spain.

[12] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat & Manuvir Das (2006): *Perracotta: Mining Temporal API Rules from Imperfect Traces*. In: *International Conference on Software Engineering*. pp. 282–291.