Mastering Change @ Runtime*

Klaus Havelund

Jet Propulsion Laboratory
California Institute of Technology
California, USA

This brief paper is a response to a call [7] for opinion statements from members of the editorial board of the upcoming journal: *LNCS Transactions on Foundations for Mastering Change (FoMaC)*. In the call it says:

> *FoMaC intends to establish a forum for formal methods-based research that fosters a discipline for rigorously dealing with the nature of today's agile system development, which is characterized by unclear premises, unforeseen change, and the need for fast reaction, in a context of hard to control frame conditions, like third party components, network-problem, and attacks.*

The phases covered span from meta modeling to modeling and design, implementation, runtime and finally evolution/migration. In the extreme, all software correctness issues can be considered as purely change issues, where the fundamental question is the following: given a program P, potentially empty, will the addition of the program fragment Δ make $P + \Delta$ satisfy a property ψ? Program fragments Δ can here be understood liberally, as for example edit commands (replace these lines of code with these lines of code), refinements - as in stepwise program refinement suggested for wide-spectrum development languages such as VDM [4], aspects - as in aspect oriented programming, plans - as in planning (the Δ is a new plan), etc. As such, in the extreme, the topic of correctness under change can be considered as the well known topic of correctness. An interesting question is: what is the connection between the concept of change as a special topic and then the more general and traditional software correctness issue?

As is well known, analysis for insurance of correctness as well as security can be performed statically (of code structure) or dynamically (of execution traces). In the realm of static analysis, version control is of course a basic very useful technology. One can imagine version control systems being brought to the next level by being integrated with static analysis tools, explicitly supporting program refinement, as well as smart IDEs, which visually highlight changes as they are made in the editor. In general, the integration of specification, programming and verification, as explored for example in DAFNY [1], as well as in the earlier VDM [4], should in principle make change easier. This includes adoption of high-level programming languages such as SCALA [2].

However, we observe that ensuring correctness of software using static methods is extremely challenging, and therefore our systems should be constructed

* 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.

to be robust in the face of errors at *runtime*. In the realm of dynamic analysis, one can distinguish between *detection* of change during runtime, and *causing* change during runtime. Detection of change can occur by monitoring a system's execution while checking its behavior against a formalized specification of expected behavior. Here a system can be considered as emitting a sequence of observable events, which are fed into a monitor, which as a second input takes a specification of expected behavior. The trace is then matched against the specification. Events in practice will carry data, and it must be possible to refer to data in specifications [5]. The specification can be written by humans, or it can be learned from nominal executions, also referred to as specification mining [6]. Properties can be expressed in various specification languages, ranging from state machines, regular expressions, temporal logics, rule-based systems to also include refinement-based notations as discussed previously.

Detection of a property violation can be used to cause a change of behavior by triggering fault-protection code, which steers the application out of a bad situation. The simplest possible fault-protection strategy is to reboot the system, a strategy which in practice is very common. At the other end of the scale is planning and scheduling techniques, which continuously adapt to the current situation. A planner, upon request, generates a new program (plan) to be executed for the next time period in order to achieve a given goal. Planning is related to program synthesis. In contrast to planning, program synthesis usually occurs before deployment and has more static nature, but in theory the two topics are closely related. The common theme for planning and synthesis is the exploration of new ways to write programs that make the correctness question, and thereby change question, less of an issue. For a survey relating verification and validation to planning and scheduling, see [3].

References

1. S. Bensalem, K. Havelund, and A. Orlandini. Verification and validation meet planning and scheduling. *Software Tools for Technology Transfer (STTT)*, 16(1):1–12, February 2014.
2. D. Bjørner and C. B. Jones. *Formal Specification and Software Development*. Prentice Hall International, 1982. ISBN 0-13-880733-7.
3. Dafny website. http://research.microsoft.com/en-us/projects/dafny.
4. K. Havelund. Monitoring with data automata. In T. Margaria and B. Steffen, editors, *6th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation, ISoLA 2014. Track: Statistical Model Checking, Past Present and Future (organized by K. Larsen and A. Legay), Corfu, Greece, October 8-11*, LNCS, this volume. Springer, 2014.
5. M. Isberner, F. Howar, and B. Steffen. Learning register automata: from languages to program structures. *Machine Learning*, 96(1-2):65–98, 2014.
6. Scala website. http://www.scala-lang.org.
7. B. Steffen. LNCS transactions on foundations for mastering change: Preliminary manifesto. In T. Margaria and B. Steffen, editors, *6th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation, ISoLA 2014, Corfu, Greece, October 8-11*, LNCS, this volume. Springer, 2014.