Static and Runtime Verification, Competitors or Friends? (Track Summary)

Dilian Gurov[1], Klaus Havelund[2(✉)], Marieke Huisman[3], and Rosemary Monahan[4]

[1] KTH Royal Institute of Technology, Stockholm, Sweden
dilian@kth.se
[2] Jet Propulsion Laboratory, Pasadena, USA
klaus.havelund@jpl.nasa.gov
[3] University of Twente, Enschede, The Netherlands
m.huisman@utwente.nl
[4] Maynooth University, Maynooth, Ireland
Rosemary.Monahan@nuim.ie

1 Motivation and Goals

Over the last years, significant progress has been made both on static and runtime program verification techniques, focusing on increasing the quality of software. Within this track, we would like to investigate how we can leverage these techniques by combining them. Questions that will be addressed are for example: what can static verification bring to runtime verification to reduce impact on execution time and memory use, and what can runtime verification bring to static verification to take over where static verification fails to either scale or provide precise results? One can to some extent consider these two views (static verification supporting runtime verification, and runtime verification supporting static verification) as fundamentally representing the same scenario: prove what can be proved statically, and dynamically analyze the rest.

The session will consist of several presentations, some on the individual techniques, and some on experiences combining the two techniques. When preparing this session, we aimed at finding a balance between static and runtime verification backgrounds of the presenters. This is also reflected by the papers associated to this track. There are several papers describing systems that first attempt to verify as much as possible by static verification, and then use runtime verification for the properties that cannot be verified statically. There is another group of papers that use static program information to generate appropriate runtime checks. Finally, a last group of papers discuss program specification techniques for static verification, and how they can be made suitable for runtime verification, or the other way round.

K. Havelund—The research performed by this author was carried out at Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

T. Margaria and B. Steffen (Eds.): ISoLA 2016, Part I, LNCS 9952, pp. 397–401, 2016.
DOI: 10.1007/978-3-319-47166-2_27

During the conference, three panel discussions on this topic are planned. The first panel focuses on static verification. What are the challenges, and how can it benefit from runtime verification? The second panel focuses on the opposite question: what are the challenges in runtime verification, and how can it benefit from static verification? The last panel discusses future research directions in this area, and what are the most promising ideas for combining static and runtime verification. Concrete topics that will be discussed include the limitations and benefits of each approach, how we can combine efforts to benefit verification, what are the overheads/benefits of combining efforts, industrial application in each area, industrial needs, etc.

2 Contributions

The paper contributions in cbelow. The papers are ordered according to the three sessions of the track: (1) how can static verification benefit from runtime verification? (2) how can runtime verification benefit from static verification? and (3) how can we bridge the gap? (more generally). The papers are ordered alphabetically according to authors within each session.

2.1 How Can Static Verification Benefit from Runtime Verification?

Ahrendt et al. [1] (*StaRVOOrS Episode II, Strengthen and Distribute the Force*) build on StaRVOOrS as presented at ISoLA 2012, which aims at a unifying framework for static and runtime verification of object-oriented software. Advances on a unified specification language for data and control oriented properties, a tool for combined static and runtime verification, and experiments are presented. Future research concern (i) the use of static verification techniques to further optimize the runtime monitor, and (ii) extending the framework to the distributed case. A roadmap for addressing these challenges is presented.

Azzopardi et al. [2] (*A Model-Based Approach to Combining Static and Dynamic Verification Techniques*) present how static and runtime verification can be used to ensure safety of systems that are to be used in an unknown context. The system developer has to provide a model of the system. This model then is used to find the appropriate context for the system to work in, and an attempt is made to statically verify the desired properties of the composed system. Any property (or part of a property) that cannot be verified statically will be verified dynamically. Moreover, it will also be verified dynamically whether the concrete implementation of the system respects the model. In some cases, knowledge about the properties that will be monitored can be used to reduce the model. The paper discusses a concrete example of this approach for an online payment ecosystem.

Bodden et al. [3] (*Information Flow Analysis for Go*) present parts of the theory and implementation of an information flow analysis of Go programs. The purpose is to detect the flow of so-called tainted values, from untrusted

sources (such as reading from input) to so-called sinks, which represent locations where such untrusted data should not end up. Go allows for concurrent programming via channels, requiring special techniques. Discussions include how dynamic analysis can be applied, to monitor execution paths, that cannot be determined safe due to the conservative static analysis. An option is to stop the execution of the program when a tainted datum is about to reach a sink. A dynamic coverage tool can also provide information as to how many of these potentially unsafe paths have been executed and verified.

2.2 How Can Runtime Verification Benefit from Static Verification?

Goodloe [5] (*Challenges in High-Assurance Runtime Verification*) first presents an overview of the Copilot RV framework, followed by several challenges that are barriers to realizing high-assurance runtime verification. More specifically, these challenges relate specification, observability of data, traceability from requirements, fault tolerance, composition of runtime verification and the system under observation, monitor specification and monitor correctness. While the challenges are formulated generally, Goodloe addresses them concretely in the context of the Copilot RV framework. Additional challenges to be addressed in future work, as well as challenges regarding the use of automated verification tools for high-assurance runtime verification, are also discussed.

Kosmatov et al. [6] (*Static versus Dynamic Verification in Why3, Frama-C and SPARK 2014*) describe the Why3 system, and two tools that use the Why3 system as a backend, namely Frama-C and SPARK. As these systems focus on different kinds of verification techniques (SPARK concentrates on runtime verification, while Frama-C and Why3 favor static verification) and properties of interest, there are differences in the specification languages, in the treatment of ghost code, and in the treatment of proof failures. The paper provides an in-depth discussion of these differences.

Reger [9] (*Considering Typestate Verification for Quantified Event Automata*) sketches how static verification techniques for type states can be used on a commonly used specification framework for runtime verification, namely quantified event automata. He gives an overview of type states and quantified event automata, and then sketches how type state techniques can be used, using some example properties specified as quantified event automata.

2.3 How Can We Bridge the Gap?

Leofante et al. [7] (*Combining Static and Runtime Methods to Achieve Safe Standing-Up for Humanoid Robots*) address how to improve a scripted stand up strategy for robots, making it safe and stable, using a combination of runtime verification and static verification. This paper describes a novel approach to achieve safe standing-up for humanoid robots. It proposes a combination of three methods. The first is reinforcement learning that uses Q-learning based on a robot simulator to construct a standing-up strategy. The second method is greedy model repair that uses efficient probabilistic model checkers to repair the

strategy to avoid given unsafe states with a given probabilistic threshold. These two methods result in an initial strategy that is deployed on the robot. As the strategy has been obtained on an idealized model of the real robot and environment, it may still not be adequate. Therefore, the third method is runtime verification with a feedback loop to observe the real-time behavior of the robot and adapt the strategy on the go. The implementation of the presented theory is ongoing, but already some experimental results for (model free) reinforcement learning strategies are presented.

Leucker [8] (*On Combinations of Static and Dynamic Analysis*) elaborates in his presentation on the similarities and differences of model checking and runtime verification, and how they can benefit from each other. In particular, if model checking an abstract version of the system fails, how can runtime verification be used to investigate the unsuccessful run? The presentation also discusses ideas for how to use information obtained by static verification to improve runtime verification results.

Eilertsen et al. [4] (*Safer Refactorings*) present a method to avoid refactorings changing the behavior of a program. Refactorings are a way to restructure a program's code. If a refactoring is wrongly applied, this might actually change the behavior of the program, which should be avoided. Eilertsen et al. propose a technique to identify when the program's behavior is actually changed. For two concrete refactorings (extract local variable, and extract and move method) they describe how this is done. Essentially, together with the refactoring they generate an assertion which will fail if the refactoring changed the program behavior. To validate their approach, they automatically apply these refactorings on a large code base, and use unit tests to identify how many assertions actually fail.

References

1. Ahrendt, W., Pace, G.J., Schneider, G.: StaRVOOrS episode II, strengthen and distribute the force. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016, Part I, LNCS, vol. 9952, pp. 402–415. Springer, Heidelberg (2016)
2. Azzopardi, S., Colombo, C., Pace, G.: A model-based approach to combining static and dynamic verification techniques. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016, Part I, LNCS, vol. 995, pp. 416–430. Springer, Heidelberg (2016)
3. Bodden, E., Pun, K.I., Steffen, M., Stolz, V., Wickert, A.-K.: Information flow analysis for go. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016, Part I, LNCS, vol. 9952, pp. 431–445. Springer, Heidelberg (2016)
4. Eilertsen, A.M., Bagge, A.H., Stolz, V.: Safer refactorings. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016, Part I, LNCS, vol. 9952, pp. 517–531. Springer, Heidelberg (2016)
5. Goodloe, A.: Challenges in high-assurance runtime verification. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016, Part I, LNCS, vol. 9952, pp. 446–460. Springer, Heidelberg (2016)
6. Kosmatov, N., Marché, C., Moy, Y., Signoles, J.: Static versus dynamic verification in Why3, Frama-C and SPARK. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016, Part I, LNCS, vol. 9952, pp. 461–478. Springer, Heidelberg (2014)

7. Leofante, F., Vuotto, S., Ábrahám, E., Tacchella, A., Jansen, N.: Combining static and runtime methods to achieve safe standing-up for humanoid robots. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016, Part I, LNCS, vol. 9952, pp. 496–514. Springer, Heidelberg (2016)
8. Leucker, M.: On combinations of static and dynamic analysis. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016, Part I, LNCS, vol. 9952, pp. 515–516. Springer, Heidelberg (2016)
9. Reger, G.: Considering typestate verification for quantified event automata. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016, Part I, LNCS, vol. 9952, pp. 479–495. Springer, Heidelberg (2016)