M. Tichy, E. Bodden, M. Kuhrmann, S. Wagner, J.-P. Steghöfer (Hrsg.): SE 2018,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2018 145

Symbolic Execution for Realizability-Checking of Scenario-based Specifications[1][2]

Joel Greenyer[3], Timo Gutjahr[4]

Abstract: Scenario-based specification with the Scenario Modeling Language (SML) is an intuitive approach for formally specifying the behavior of reactive systems. SML is close to how humans conceive and communicate requirements, yet SML is executable and simulation and formal realizability checking can find specification flaws early. The realizability checking complexity is, however, exponential in the number of scenarios and variables. Therefore algorithms relying on explicit-state exploration do not scale and, especially when specifications have message parameters and variables over large domains, fail to unfold their potential. In this paper, we present a technique for the symbolic execution of SML specifications that interprets integer message parameters and variables symbolically. It can be used for symbolic realizability checking and interactive symbolic simulation. We implemented the technique in ScenarioTools. Evaluation shows drastic performance improvements over the explicit-state approach for a range of examples. Moreover, symbolic checking produces more concise counter examples, which eases the comprehension of specification flaws.

Keywords: Reactive Systems; Scenario-Based Modeling; Realizability Checking; Symbolic Execution

Many software-intensive systems, especially cyber-physical systems, consist of reactive components that interact with each other and the environment in order to realize complex and often safety-critical functionality. Scenario-based modeling with *Live Sequence Charts* (LSCs) [HM03], and a textual variant, the *Scenario Modeling Language* (SML), is an intuitive, yet formal approach for specifying the behavior of such systems. SML extends LSCs with concepts for specifying environment assumptions and dynamic topologies [Gr17].

LSC/SML specifications are executable via the *play-out* algorithm [HM03], and can be analyzed via simulation. Violations encountered during simulation runs hint at possible specification flaws. Simulation alone, however, cannot prove the absence of flaws. For this purpose, there exist methods for proving the *realizability* of LSC/SML specifications [BH05, MS12], i.e., checking whether there exist an implementation of the specification or not, due to contradicting requirements or other inconsistencies.

[1] This is an extended abstract of [GG17]

[2] Funded by the German Israeli Foundation for Research and Development (GIF), grant No. 1258, and the Deutsche Forschungsgemeinschaft (DFG), project EffiSynth.

[3] Leibniz Universität Hannover, Software Engineering Group, Welfengarten 1, 30167 Hannover, Germany, greenyer@inf.uni-hannover.de

[4] Zühlke Engineering GmbH, Podbielskistraße 333, 30659 Hannover, Germany Timo.Gutjahr@zuehlke.com

These approaches, however, usually do not scale well, since they rely on an exploration of the state space induced by the specification, which can be exponential in the number of scenarios and variables in the specification. Approaches relying on BDD-based algorithms [MS12] only support basic scenario-based language concepts, since the mapping of elaborate scenario language concepts to lower-level formalisms and tools is very difficult.

We therefore investigated how to extend the existing play-out algorithms with *symbolic execution* [Ki76]. Symbolic execution, known from program analysis, executes a program with symbolic values as inputs; it can deduce for which constraints on these inputs it is possible to arrive at a particular part of a program that is of interest, e.g., the violation of an assertion. In the case of symbolic play-out, environment event parameters and initial component state attributes are interpreted symbolically. The particular challenge, compared to the symbolic execution of sequential programs, is twofold: (1) during a symbolic play-out execution, inputs occur repeatedly, and (2) in an execution state, multiple scenarios can be active and formulate conditions over message parameter and variable values. One step in symbolic play-out could thus result in branching into as many paths as required to cover all possible combinations of branchings implied by each active scenario.

We formalized symbolic play-out, and implemented it within ScenarioTools, which we combined with Z3 for constraint solving. We experimented with different strategies to match symbolic execution states, which is required for systematically exploring execution paths. The evaluation shows that symbolic play-out can drastically improve realizability checking performance for specifications with message parameters and variables over large domains.

References

[BH05] Bontemps, Yves; Heymans, Patrick: From Live Sequence Charts to State Machines and Back: A Guided Tour. IEEE Transactions on Software Engineering, 31(12):999–1014, 2005.

[GG17] Greenyer, Joel; Gutjahr, Timo: Symbolic Execution for Realizability-Checking of Scenario-Based Specifications. In: 2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS). volume 00, pp. 312–322, Sept. 2017.

[Gr17] Greenyer, Joel; Gritzner, Daniel; Gutjahr, Timo; König, Florian; Glade, Nils; Marron, Assaf; Katz, Guy: ScenarioTools – A tool suite for the scenario-based modeling and analysis of reactive systems. Science of Computer Programming, 149(Supplement C):15 – 27, 2017. Special Issue on MODELS'16.

[HM03] Harel, D.; Marelly, R.: Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine. Springer, 2003.

[Ki76] King, James C.: Symbolic execution and program testing. Communications of the ACM, 19(7):385–394, 1976.

[MS12] Maoz, Shahar; Sa'ar, Yaniv: Assume-Guarantee Scenarios: Semantics and Synthesis. In (France, R. B.; Kazmeier, J.; Breu, R.; Atkinson, C., eds): Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012. Proceedings. volume 7590 of Lecture Notes in Computer Science. Springer, pp. 335–351, 2012.