Jens Knoop, Uwe Zdun (Hrsg.): Software Engineering 2016,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2016 79

A Primer on Counterexample Guided Abstraction Refinement of Product-Line Behavioural Models

Maxime Cordy[1], Bruno Dawagne[2], Patrick Heymans[3], Axel Legay[4], Martin Leucker[5] and Pierre-Yves Schobbens[6]

Abstract: The model-checking problem for Software Products Lines (SPLs) is harder than for single systems: variability constitutes a new source of complexity that exacerbates the state-explosion problem. Abstraction techniques have successfully alleviated state explosion in single-system models. However, they need to be adapted to SPLs, to take into account the set of variants that produce a counterexample. In this paper, we recall the main ideas of a paper published elsewhere that applies CEGAR (Counterexample-Guided Abstraction Refinement) and desings new forms of abstraction specifically for SPLs. Experiments are carried out to evaluate the efficiency of our new abstractions. The results show that our abstractions, combined with an appropriate refinement strategy, hold the potential to achieve large reductions in verification time, although they sometimes perform worse.

Keywords: Software Product Lines, Model Checking, CEGAR, Abstraction

Summary[7]

Software Product Lines (SPLs) are families of similar software systems developed together. SPL engineering aims to facilitate the development of the members of a family (called products or variants) by identifying upfront their commonalities and differences. Variability in SPLs is commonly represented in terms of features, i.e., units of difference between products that appear natural to stakeholders. The emergence and the increasing popularity of SPLs have raised the need for SPL-specific quality assurance techniques. Indeed, engineers have to provide solid evidence that all the products they build satisfy their intended requirements. Moreover, in case of failure, they should identify which features, or combinations of features, are responsible for the errors in order to facilitate repair.

Model checking is an automated technique to verify a behavioural model of a system against a property expressed in temporal logic. It relies on an exhaustive exploration of the model in search for counterexamples, i.e., executions that violate the property to verify. Due to its exhaustiveness, model checking is costly in time and memory. When applied to real systems with a typically huge state space, model checking faces a combinatorial blow-up called state explosion. The model-checking problem is even harder for SPLs: in this case, the model checker must either prove the absence of errors or find a counterexample for each variant that can produce a violation. Given that the worst-case number of products of an SPL is exponential in the number of features, variability dramatically exacerbates state explosion. As a consequence, it is not feasible to apply single-system model

[1] University of Namur, Belgium, mcr@info.fundp.ac.be [2] University of Namur, Belgium, bdawagne@student.fundp.ac.be [3] University of Namur, Belgium, phe@info.fundp.ac.be [4] INRIA Rennes, France, axel.legay@inria.fr [5] University of Lübeck, Germany, leucker@isp.uni-luebeck.de [6] University of Namur, Belgium, pys@info.fundp.ac.be
[7] This paper summarizes the paper published in [Co14]. References and further details can be found there.

checking to the thousands of variants that can compose real-world SPLs. In recent years, many variability-aware techniques have been designed to address the SPL model checking problem. These techniques keep track of variability information contained in an SPL behavioural model to associate each execution path to the exact set of variants able to produce it. By doing so, they are able to identify the set of products that violate a given property, and to report a counterexample of violation for each of them. Moreover, being aware of variability allows them to check behaviour common to several products only once. One of the most effective answers to state explosion is model abstraction, which creates more concise and therefore easier to verify models of the system, typically by merging similar states. This reduced size often comes at the cost of inaccuracies in the models: A reported counterexample can therefore be spurious, that is, it exists within the abstract model but the not in the real, concrete model. In this case, the abstraction must be refined to eliminate this false positive. Common methods to achieve this refinement make use of the spurious counterexample itself. They give rise to Counterexample Guided Abstraction Refinement (CEGAR), i.e. abstraction techniques that iteratively refine an abstract model until either they find a real counterexample or they can prove the absence of violation.

In spite of their success in single-system model checking, abstraction techniques for SPLs have received little attention. In [Co14], this gap is filled by proposing SPL-specific abstraction procedures based on CEGAR. Applying CEGAR to SPLs is more tedious because a counterexample can be real for some products and spurious for others. This observation leads to two refinement strategies: one refines the model as soon as it finds a spurious counterexample, whereas the other performs the spuriousity check and the refinement after the discovery of all counterexamples. As for the abstraction of the model, we distinguish between (1) state abstraction that only merge states as in single-model abstraction, (2) feature abstraction that modifies only the variability information contained in the model, and (3) mixed abstraction that combines the previous two types. This latter type is the most complicated to implement, as spuriousness can originate from the merging of states, the abstraction of features, or both. In [Co14], the correctness of the approach is proven on the basis of mathematical relations such as simulation relations. Morover, both abstractions and their combination were implemented in ProVeLines, an SPL model checker previously developed by some of the authors. Experiments were carried out to evaluate the efficiency of different combinations of refinement strategies and abstractions. The results tend to show that state abstraction brings performance gains most of the time, whereas feature abstraction generally results in small losses of performance but achieve huge decreases of verification time in some cases. Preliminary experiments on mixed abstraction tend to show that its performance is comparable to that of state abstraction, although slightly worse on average. Other abstractions of this kind could, however, be designed as part of future work and yield better results

References

[Co14] Cordy, Maxime; Dawagne, Bruno; Heymans, Patrick; Legay, Axel; Leucker, Martin; Schobbens, Pierre-Yves: Counterexample Guided Abstraction Refinement of Product-Line Behavioural Models. In: FSE'14. ACM, Hong Kong, China, November 2014.