On the Probabilistic Symbolic Analysis of Programs

Antonio Filieri
University of Stuttgart, Germany
filieri@informatik.uni-stuttgart.de

Corina S. Păsăreanu
Carnegie Mellon Silicon Valley, NASA Ames, USA
corina.s.pasareanu@nasa.gov

Abstract

Recently we have proposed symbolic execution techniques for the probabilistic analysis of programs. These techniques seek to quantify the probability of a program to satisfy a property of interest under a relevant usage profile. We describe recent advances in probabilistic symbolic analysis including handling of complex floating-point constraints and nondeterminism, and the use of statistical techniques for increased scalability.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification

Keywords Probabilistic Symbolic Execution, Nondeterminism

1. Probabilistic Symbolic Analysis

Probabilistic software analysis aims at quantifying the probability that a software system satisfies a given property. This analysis has been traditionally performed through probabilistic model checking [1] and usually applied on architecture-level abstractions of the software. This allows performing quantitative analysis during early stages of design, but not, in general, on code artifacts. In the last few years, Probabilistic Symbolic Execution (PSE) [2, 5, 6, 11] has been proposed to bring probabilistic analysis directly at code level, with a promising applicability scope.

Symbolic Execution [8] is a program analysis technique that executes programs on unspecified inputs, by using symbolic inputs instead of concrete data. The state of a symbolically executed program is defined by the (symbolic) values of program variables, a *path condition* (PC), and a program counter. The PC is a (quantifier-free) boolean formula over the symbolic inputs; it accumulates constraints that concrete inputs must satisfy for an execution to follow the associated path. The program counter identifies the next statement to execute. A symbolic *execution tree* characterizes the execution paths followed during the symbolic execution of a program. The nodes of the tree represent program states and the arcs the transitions between states due to the execution of an instruction. To deal with non termination, we limit the length of executions.

Probabilistic Symbolic Execution enhances symbolic execution allowing to quantify the probability for a PC to be satisfied by an input value, according to a given usage profile (a different though equivalent definition is given in [11]). This new feature can be exploited to perform multiple types of analysis, including: computing the probability of a program to satisfy a property ϕ as the sum of the probabilities of satisfying the PCs associated to execution paths leading to the satisfaction of ϕ; assessing the relevance of an execution path for the behavior of the software under the current usage profile (for example, to evaluate the quality of a test suite or to drive improvement/refactoring decisions).

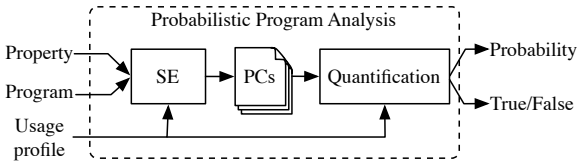

Figure 1. Probabilistic program analysis chain.

Figure 1 shows the basic workflow of probabilistic program analysis, as introduced in [5]. The inputs are the source code of the program, the usage profile, and a target property. The program is first executed symbolically to collect the PCs of the execution paths leading to the satisfaction of the target property. Then, the solution space of these PCs is quantified in order to either compute the probability for the property to be satisfied under the provided usage profile, or to verify such probability meets prescribed constraints. Notice that the usage profile is used to both prune out unfeasible execution paths during symbolic execution and to compute the probability of satisfying the target property.

2. The Quantification Problem

Quantifying the probability of satisfying a path condition, under a given usage profile, is in general a hard task. If the input domain is a bounded, albeit very large, subset of the integer numbers, and the constraints appearing in the PCs are limited to linear algebraic ones (or over restricted classes of polynomials), efficient counting techniques have been developed, based on Barvinok's algorithm [4]. For linear algebraic constraints over floating-point numbers, a bounding algorithm has been proposed in [11] to compute arbitrarily small intervals containing the actual probability of satisfying the target property. Arbitrarily complex finitized data structures can in principle be handled with Korat [5], which performs a "smart" enumeration of the instances satisfying complex constraints. However, despite its generality, such enumeration may still be infeasible for large domains [5]. However none of these techniques can deal with non-linear constraints over floating-point domains.

To deal with arbitrarily complex numerical constraints, in [2] we proposed a compositional approximate quantification method, suitable for both integers and floating-point numbers, built upon Monte Carlo integration. The latter is notoriously one of the most broadly applicable methods, since based on simulation. However, the complexity of simulation might be overwhelming, requiring

CONF 'yy, Month d–d, 20yy, City, ST, Country.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

an intractable amount of time to quantify over large domains and complex constraints. Furthermore, to achieve a reasonable accuracy a very large number of samples might be required, especially when dealing with fairly irregular functions like the boolean functions evaluating complex conditions. To handle these limitations, we improved naive Monte Carlo integration in two directions.

The first direction targets mostly efficiency and is a divide and conquer strategy aiming at identify independent clauses in a PC that can be analyzed independently, similarly to [5]. For each clause the result of integration is estimated, as well as its accuracy in terms of the variance of the corresponding estimator. The local results are then combined according to analytical composition rules so to obtain a global estimate for the integration of the whole PC. The local uncertainty is propagated as well, producing a conservative assessment of the quality of the results. The outcome is twofold. First, the local results for independent clauses can be heavily reused [2, 5] with a significant speedup especially for large programs. Second, the independent clauses usually predicate only on a subset of the program variables. This reduces the dimensionality of the problem resulting in faster and more accurate local integrations.

The second targets mostly accuracy and involves the use of an Interval Constraints Solver (ICP) [7] to partition the integration domain into a finite set of boxes. If a box contains no solutions, the result of integration, restricted to that box, will be constantly 0. Analogously, if all the points in a box are solutions of the constraint under analysis, the result of integration will be the volume of the box. In these two cases the variance of the Monte Carlo estimators within the boxes is 0, meaning the result has no uncertainty. For the other boxes, a non-zero variance will be returned. The results of the various boxes can be combined, as specified in statistic for *stratified sampling*. Depending on the ability of ICP to identify boxes containing either only or no solutions, the accuracy of integration may significantly increase. This is often the case for Software [2].

3. Dealing with Nondeterminism

The execution of nondeterministic programs does not depend only on its inputs but also on the used *scheduler*, i.e. the component deciding which *actions* to take to resolve nondeterministic choices (e.g. thread scheduling). Fixed a scheduler, execution is completely characterized by its symbolic execution tree.

In [5], we dealt with nondeterminism by computing the best (worst) linear *schedule* (i.e. ordered sequence of thread choices), which maximizes (minimizes) the probability of the program to satisfy a target property ϕ, for a given usage profile. However, linear schedules are in general not enough to design complete schedulers for an optimal execution of the program and tree-like schedulers are needed.

In later work [10], we investigate probabilistic software analysis for nondeterministic programs. In particular, we tackle the problem of synthesizing an optimal scheduler providing formal guarantees on the probability of satisfying ϕ. This scheduler is a complete tree-shaped probabilistic decision maker, i.e. it assigns a probability of being selected to all the valid alternatives for resolving a nondeterministic choice. This generalizes linear schedulers from [5] and follows classical work on Markov Decision Processes (MDPs) [1, 9] but adapted to the setting of analysis of programs, not models, and tailored to the tree shape of the symbolic .

In [10] we study exact and approximate algorithms for the problem, combining the peculiarity of PSE, sampling, and reinforcement learning to effectively trade scalability and accuracy off. We show experimentally that we significantly improve over the state-of-the-art statistical model checking for MDPs from [3].

4. Current and Future Plans

We are currently working on both introducing more efficient quantification procedures for common types of constraints (including strings and complex data structures) and on providing a better support for special kinds of usage profiles, including stateful ones, needed for example to analyze reactive systems. We are also investigating in depth statistical techniques as a way of improving the scalability of our approach.

Acknowledgments

The research described in this work has been conducted in collaboration with Mateus Borges, Marcelo D'Amorin, Matthew Dwyer, Kasper Luckow, and Willem Visser.

References

[1] C. Baier, J.-P. Katoen, et al. *Principles of model checking*. MIT press Cambridge, 2008.

[2] M. Borges, A. Filieri, M. d'Amorim, C. S. Păsăreanu, and W. Visser. Compositional solution space quantification for probabilistic software analysis. In *PLDI*, 2014 – to appear.

[3] E. Clarke and P. Zuliani. Statistical model checking for cyber-physical systems. In *ATVA*, pages 1–12, Oct. 2011.

[4] J. A. De Loera, R. Hemmecke, J. Tauzer, and R. Yoshida. Effective lattice point counting in rational convex polytopes. *Journal of Symbolic Computation*, 38(4):1273–1302, Oct. 2004.

[5] A. Filieri, C. S. Păsăreanu, and W. Visser. Reliability analysis in Symbolic PathFinder. In *ICSE*, pages 622–631, July 2013.

[6] J. Geldenhuys, M. B. Dwyer, and W. Visser. Probabilistic symbolic execution. In *ISSTA*, pages 166–176, July 2012.

[7] L. Granvilliers and F. Benhamou. Algorithm 852: Realpaver: an interval solver using constraint satisfaction techniques. *ACM Transactions on Mathematical Software*, 32:138–156, 2006.

[8] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[9] A. Kolobov. *Planning with Markov Decision Processes: An AI Perspective*. Morgan and Claypool, 2012.

[10] K. Luckow, C. S. Păsăreanu, M. Dwyer, A. Filieri, and W. Visser. Probabilistic symbolic execution for nondeterministic programs. In *CAV*, 2014 – submitted.

[11] S. Sankaranarayanan, A. Chakarov, and S. Gulwani. Static analysis for probabilistic programs: inferring whole program properties from finitely many paths. In *PLDI*, pages 447–458, June 2013.