Quantification of Software Changes through Probabilistic Symbolic Execution

Antonio Filieri¹, Corina S. Păsăreanu², and Guowei Yang³

1 University of Stuttgart, Stuttgart, Germany
2 Carnegie Mellon Silicon Valley, NASA Ames, Moffet Field, CA, USA
3 Texas State University, San Marcos, TX, USA

Abstract

Characterizing software changes is a fundamental component of software maintenance. Despite being widely used and computationally efficient, techniques that characterize syntactic program changes lack an insight on the changed program behaviors and can possibly lead to unnecessary maintenance efforts. Recent promising techniques use program analysis to produce a behavioral characterization of program changes, see e.g. [10, 12]. Behaviors are either abstracted through operational models (e.g., transition systems) or summarized through a set of logical formulae satisfied by the input-output relation (e.g., pre- and post-conditions). Checking the implication or the equivalence between the abstraction of different program versions provides a qualitative assessment of the preservation of desired behaviors or the elimination of undesired behaviors. Nonetheless, such qualitative assessment provides only true-false answers, providing limited guidance on “how far” two versions are different from one another. Recent work [8, 9] provides more informative but still only qualitative representation of the difference. We argue that a complementary quantitative representation for software changes is needed, particularly for programs required to operate under uncertainty usage profiles, where the goal of maintenance is to improve the average quality of the program instead of its worst-case performance.

In this work, we propose to compute a precise numeric characterization of a program change by quantifying the likelihood of reaching program events of interest (e.g. successful termination or assertion violations) and how that evolves in time, with each program version. Furthermore, our approach quantifies the percentage of inputs that are affected by each change. Such precise characterization of behavioral changes can be used to rank different program versions based on the execution probability of the changes and their impact on the probability of satisfying or failing desirable properties, under an uncertain usage profile.

With this new quantitative approach we are able state not only that the program has changed with a logical delta, as in the previous qualitative approaches, but we can also compute that delta affects say 30% of the program inputs, giving a clear, measurable indication for the effort necessary to re-test the program modifications.

Furthermore, after fixing a bug, existing qualitative techniques can only assess whether the new version is free of errors or not, which may be too restrictive for most realistic applications. Instead, with our approach we can quantify the probability of reaching an error in the old and new versions, expecting it to decrease with each new bug fix. In yet another scenario, consider the case of multiple candidate repairs for a given bug: our technique can be used to automatically rank them according to their probability of execution or the overall probability of failure.

The approach extends probabilistic symbolic execution [6, 4] to compute the symbolic constraints that characterize program paths in different program versions. Solution space quantification techniques [3, 1] over the collected constraints are used to precisely quantify the percentage of inputs leading to the occurrence of a target event that are affected by a change (approximate quantification with probabilistic precision guarantees has also been explored to cope with larger programs [7, 5]). Furthermore, our approach exploits the fact that program versions are largely similar to reduce cost and improve the precision of analysis by storing and reusing partial
Quantification of Software Changes through Probabilistic Symbolic Execution

analysis results from previous versions [13]. Our quantitative measures are different from simulation distances [2] which are real-valued functions between two high-level models (a specification and an implementation), computed using quantitative simulation games. In contrast we focus on different versions of the same system, analyzing directly code (not high-level models), using probabilistic techniques.

We have implemented our approach in the Symbolic PathFinder tool [11] and performed an exploratory study that considers version histories of Java code bases, under common software maintenance scenarios such as evaluating different program repairs, performing refactoring, or evaluating different mutants used in testing. The preliminary results show a promising application scope for our technique.

References


