Employing Run-time Static Analysis to Improve Concolic Execution

Position Paper

Maarten Vandercammen
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
Maarten.Vandercammen@vub.be

Coen De Roover
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
Coen.De.Roover@vub.be

***Abstract*—Dynamic symbolic execution, or *concolic execution*, is a program testing technique that systematically executes a program with the aim of exploring all feasible program paths, and locating and reporting all errors encountered in these paths. However, as the complexity of the program grows, the number of program paths explodes, making it infeasible for concolic testers to explore all of them. To reduce the number of paths to explore, several concolic testing tools have recently started employing static analysis to prune paths guaranteed by the static analysis to be safe. The concolic tester must then only focus on those paths that *might* contain an error, as reported by the analysis. However, due to imprecisions in the analysis' result, the reported errors may just be false positives, and it is up to the tester to verify whether a reported alarm is an actual error or merely a false positive. In this position paper, we propose to increase the precision of these analyses by not only performing an initial static analysis before starting concolic testing of the program, but also by launching incremental static analyses over the program at run time, and incorporating into the analyses run-time information observed by the tester. The increased precision that results from incorporating such run-time information should enable further pruning of the program paths that must be explored by the concolic tester.**

***Index Terms*—Concolic Testing, Static Analysis, Blended Analysis**

I. Introduction

In recent years, there has been an interest in blending static and dynamic analyses to combine the best of both worlds. The results of a sound static analysis over-approximate all possible program executions, at the cost of precision. The results of a dynamic analysis are precise, yet only valid for a subset of all program executions. In this position paper, we propose a novel mechanism to blend an abstract interpretation-based static analysis [5] with a dynamic analysis in the form of a concolic tester [9] [10]. The proposed blended technique employs an initial static analysis, performed before starting concolic execution of the program, and thereafter launches static analyses at run time over the program. The run-time analyses incorporate observed run-time information to increase the precision of the analysis *with respect to the program execution from which the analysis was launched*.

The remainder of this paper is organized as follows. Section II introduces the concept of run-time static analysis. Section III gives an introduction to concolic execution and explains how a concolic tester may benefit from (run-time) static analysis. Section IV briefly describes an implementation of a prototype of this technique. Section V concludes.

II. Run-time Static Analysis

Our proposed technique focuses on the *abstracting abstract machines* (AAM) technique, developed by Van Horn and Might [13], where an abstract interpreter is systematically derived from a concrete interpreter represented as a 'concrete' abstract machine written in a small-step style. AAM systematically threads all unbounded components of the concrete machine through the machine's store and limits the amount of store addresses that can be allocated. As a result, the resulting 'abstract' abstract machine, or static analysis, explores a finite number of program states during its evaluation of a program, thus ensuring that analysis terminates.

Our run-time static analysis technique builds on the close relationship between the interpreter and its derived static analysis. All run-time information on the program's execution is encoded in the concrete machine's program state. This information is made available to the analysis by abstracting the program state in a similar manner as the abstraction of the machine itself. In general, this technique therefore first involves executing a program and observing the interpreter's concrete program state. When an external client decides to commence static analysis, the concrete state is abstracted, and the abstract interpreter is initialized with this abstract state and runs until it reaches a fixpoint. Figure 1 illustrates how the run-time information enables *refining* static analysis. As the predicate depicted in Figure 1a involves a variable with a random value, the flow graph (1b) computed by a traditional static analysis must contain both branches. However, should our run-time static analysis technique be launched after binding the variable `r` to its evaluated value, and before evaluating the predicate's value, the run-time analysis has the variable's value available, and can produce a flow graph (1c) that precisely predicts the

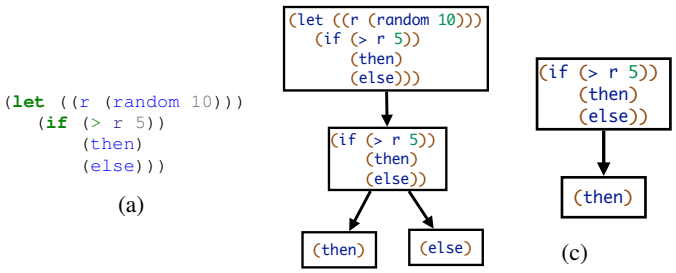

Fig. 1: A Scheme program to be analyzed (a), its initial (b), and its refined flow graph (c).

branch to be be taken. Such a run-time analysis only soundly predicts the future execution *given the current execution of the program*. Should the program be restarted, a different value for `r` may be generated, thus invalidating the result of the run-time analysis.

III. Concolic Testing

A concolic tester iteratively explores all feasible paths in a program by manipulating the values of input parameters to the program, such as program arguments, random numbers, values that are read from files etc. [3] [1].

In the first iteration, the tester assigns a random value to these input parameters while simultaneously collecting all conditional predicates it encounters into a *path constraint* that symbolically represents the conditions that must be true in order for the program to execute that particular path. After completing the first run of the program, a part of the path constraint is negated and the resulting path constraint is given to a SMT solver. If the solver can find some value for the input parameters so that this new path constraint is satisfied, concolic testing continues by restarting the program and assigning these newly computed values to the input parameters. This process continues until all feasible program paths have been explored, or until the tester exceeds some given time constraint.

We illustrate the working of a concolic tester in more detail via the example Scheme program depicted in Figure 2. In this program, the variables `a` and `b` are considered input parameters. Hence, they are symbolically represented as the input variables a_0 and b_0. Suppose now that in the first iteration, the concolic tester randomly assigns `67` to `a` and `31` to `b`. These values cause the concolic tester to execute the program statement on line 8, at which point the tester reports the error encountered there. Simultaneously with this concrete execution, the tester also collects the symbolic representation of the conditional predicates that were encountered at line 5 and 6, into a so-called *path constraint*, i.e., $a_0 > 50 \wedge b_0 <= a_0 + b_0$. After completing this run, the tester attempts to explore another path, such as the path leading to the program statement at line 7. To do this, the solver negates the last conditional predicate in the path constraint, resulting in the constraint $a_0 > 50 \wedge b_0 > a_0 + b_0$ and feeds this resulting path to a SMT solver, which subsequently attempts to assign values to a_0 and

```
1  (let* ((a (random 100))
2         (b (random 100))
3         (c (+ a b))
4         (d (+ 30 20)))
5    (if (> a d)
6        (if (> b c))
7            "ok"
8            (error))
9        "ok"))
```

Fig. 2: An example Scheme program.

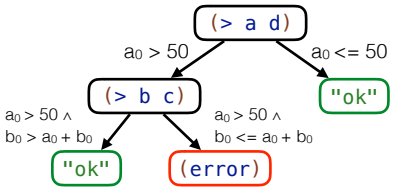

Fig. 3: The symbolic execution tree of the program depicted in Figure 2.

b_0 so that the resulting path constraint is true. Note that, as a_0 is guaranteed to be a positive number, the second conjunct $b_0 > a_0 + b_0$ can never be true. Hence no values for a_0 and b_0 can be found so that the constraint is satisfied. The concolic tester thus concludes that the constraint is unsatisfiable and that the program statement at line 7 is unreachable. At this point, the concolic tester backtracks further into the execution and tries to negate the first conjunct in the path constraint, resulting in the new constraint $a_0 <= 50$, and attempts to solve this constraint, for example by assigning the value `0` to a_0. The concolic tester re-executes the program and assigns the value `0` to `a`. It completes the program by reaching the expression at line 9. As no new branches were encountered by the tester during this last run, the tester concludes that it has explored all feasible program paths and concolic testing terminates. The three path constraints that were constructed can be collected in a symbolic execution tree that represents all possible executions of the program, as shown in Figure 3.

A. Combining Static Analysis and Concolic Testing

The number of program paths is exponential in the number of conditional branches of the program. As it is therefore often infeasible for concolic testers to explore more than a fraction of all possible program paths in a real-world application, testers employ sophisticated search strategies to select a program path that can be explored and that is most likely to result in a non-trivial error [2]. Nevertheless, in practical applications, concolic testers are not guaranteed to identify all errors located in a program. Recently, some concolic testers [8], [6], [4], [7] have started employing static analysis to greatly prune the number of program paths that must be explored by the analysis. As a sound static analysis safely proves the *absence* of errors, any program path not reported to contain an error can be avoided by the tester. However, errors that were reported by the static analysis may actually

```
1  (define (f) (error))
2  (define (g) "ok")
3  (let ((a (random 10))
4        (b (random 10))
5        (h f))
6    (if (> a 5)
7        (begin (set! h g)
8               (if (> b 5)
9                   (f)    ; error
10                  (h))) ; ok
11       "ok"))
```

Fig. 4: Another example Scheme program.

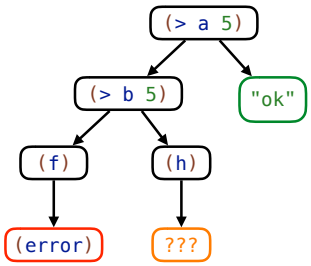

Fig. 5: The initial abstract state graph of the program depicted in Figure 4.

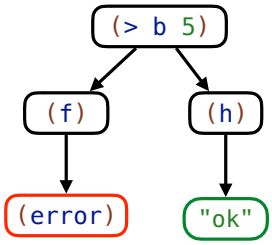

Fig. 6: The abstract state graph resulting from run-time static analysis of the same program.

be false positives that are the result of imprecision in the static analysis. The tester therefore exercises these paths in order to verify whether the reported error is an actual error. Note that the tester relies on the static analysis to be sound: as the tester only explores program paths highlighted by the analysis, a false negative produced by the analysis would not be caught by the tester.

Consider the program listed in Figure 4 and its abstract state graph, produced by performing a simple initial static analysis over this program before starting concolic testing, in Figure 5. It can be seen from the state graph that the analysis considers the path where `a` is not larger than 5 to be safe, the path where both `a` and `b` are larger than 5 to be unsafe, and the path where `a` is larger than 5 but `b` is not to be *potentially* unsafe. The conclusion for this last path is due to imprecision that arose in the static analysis: if we employ e.g., a very simple-but-fast constant propagation analysis that does not precisely track reassignments to variables, such as the reassignment to the function `h` at line 7, then the analysis cannot conclude that calling `h` at line 10 is safe. As a result, any concolic tester relying on this analysis can only prune the first path and must still explore the other two paths.

B. Combining Run-time Static Analysis and Concolic Testing

The safe path where `h` is called can be pruned by employing a run-time static analysis, in combination with an initial static analysis performed before starting testing of the program. If the tester starts exploring the error path that results in the call to `f` at line 9, the tester could launch a run-time static analysis at line 7, after the reassignment to `h` but before the paths to the calls of `f` and `h` are split. At this point, the run-time analysis can incorporate the knowledge that `h` equals the safe function `g`, thereby avoiding the imprecision caused by the reassignment of `h` at line 7. In this case, the abstract state graph produced by the run-time analysis, as depicted in Figure 6, is precise enough to determine that calling `h` will not result in an error, hence this path should not be explored by the tester. In general, a run-time analysis could be launched over the program when the tester has reached a common ancestor node to two or more paths that are potentially unsafe, in the hope that the run-time information available at that point enables the static analysis to be precise enough to determine whether or not any of the potentially unsafe paths can be pruned.

IV. Implementation and Future Work

We have implemented a prototype of this technique[1] in the Scala-AM framework [12], [11] to test Scheme programs.We are currently in the process of extending and improving the prototype and evaluating its effectiveness. To overcome the performance overhead incurred by performing (potentially multiple) static analyses over the program at run time, we aim to make these analyses incremental with respect to each other, so that one analysis can reuse part of the work computed by another. Furthermore, we expect this technique to be most effective when it employs simple-but-fast analyses that have a low execution time but a high degree of imprecision, as these imprecisions can hopefully be resolved by incorporating run-time information.

V. Conclusion

We have proposed a novel technique for improving precision of a static analysis at run time, using run-time information on the program's execution. We propose to couple this technique to a concolic tester, to reduce the number of program paths that the tester must explore.

References

[1] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *arXiv preprint arXiv:1610.00502*, 2016.

[2] Jacob Burnim and Koushik Sen. Heuristics for scalable dynamic test generation. In *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, pages 443–446. IEEE, 2008.

[3] Ting Chen, Xiao-song Zhang, Shi-ze Guo, Hong-yuan Li, and Yue Wu. State of the art: Dynamic symbolic execution for automated test generation. *Future Generation Computer Systems*, 29(7):1758–1773, 2013.

[4] Maria Christakis, Peter Müller, and Valentin Wüstholz. Guiding dynamic symbolic execution toward unverified program executions. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 144–155, New York, NY, USA, 2016. ACM.

[5] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, 1977.

[1] Available at https://github.com/mvdcamme/scala-am/tree/concolic_prototype

[6] Christoph Csallner and Yannis Smaragdakis. Check 'n' crash: Combining static checking and testing. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 422–431, New York, NY, USA, 2005. ACM.

[7] Kostas Ferles, Valentin Wüstholz, Maria Christakis, and Isil Dillig. Failure-directed program trimming. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 174–185, New York, NY, USA, 2017. ACM.

[8] Xi Ge, Kunal Taneja, Tao Xie, and Nikolai Tillmann. Dyta: Dynamic symbolic execution guided with static verification results. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 992–994, New York, NY, USA, 2011. ACM.

[9] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, June 2005.

[10] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[11] Quentin Stiévenart, Jens Nicolay, Wolfgang De Meuter, and Coen De Roover. Building a modular static analysis framework in scala (tool paper). In *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala*, SCALA 2016, pages 105–109, New York, NY, USA, 2016. ACM.

[12] Quentin Stiévenart, Maarten Vandercammen, Wolfgang De Meuter, and Coen De Roover. Scala-am: A modular static analysis framework. In *Source Code Analysis and Manipulation (SCAM), 2016 IEEE 16th International Working Conference on*, pages 85–90. IEEE, 2016.

[13] David Van Horn and Matthew Might. Abstracting abstract machines. *SIGPLAN Not.*, 45(9):51–62, September 2010.