A Late Treatment of C Precondition in Dynamic Symbolic Execution Testing Tools

Mickaël Delahaye[1] and Nikolai Kosmatov[2]

[1] Université Grenoble Alpes, LIG, 38041 Grenoble, France
firstname.lastname@imag.fr
[2] CEA, LIST, Software Reliability Laboratory, PC 174, 91191 Gif-sur-Yvette, France
firstname.lastname@cea.fr

Abstract. This paper presents a novel technique for handling a precondition in dynamic symbolic execution (DSE) testing tools. It delays precondition constraints until the end of the program path evaluation. This method allows PathCrawler, a DSE tool for C programs, to accept a precondition defined as a C function. It provides a convenient way to express a precondition even for developers who are not familiar with specification formalisms. Our initial experiments show that it is more efficient than early precondition treatment, and has a limited overhead compared to a native treatment of a precondition directly expressed in constraints. It has also proven useful for combinations of static and dynamic analysis.

Keywords: test input generation, dynamic symbolic execution, concolic testing, executable preconditions.

1 Introduction

In software testing, the *precondition* of the program under test (often called *test context*) specifies the admissible values of program inputs on which the program is supposed to be executed and should be tested. This ability to select test input domains is essential to test generation. Indeed, it allows concentrating the testing effort on admissible inputs of the program. The most common use of the test precondition is to select inputs for which the behavior of the program is specified. In that particular case, the test precondition corresponds to the specification precondition. Other uses include testing outside the specification to check for unwanted behaviors and partitioning the test domain.

In the case of automated test input generation tools, the precondition offers two interesting challenges. First, one must encode the precondition in a formalism understood by the tool. Second, the tool must take into account the precondition in its test generation process to minimize rejects for test inputs outside the precondition.

PATHCRAWLER [1] is a test input generation tool for C programs. It is based on *dynamic symbolic execution* (DSE), a technique that combines concrete execution and symbolic execution of the program under test. Originally PATHCRAWLER accepted a precondition written in a declarative constraint-based formalism specific to the tool, referred below as *native precondition*. But user feedback encouraged us to find an alternative solution.

In this paper, we propose a new approach to handle the precondition in a DSE tool, written in the tested language. It is based on a late exploration of the precondition's code during the test generation. This paper also provides an experimental evaluation of the late-precondition technique implemented in PATHCRAWLER. Our approach offers a greater expressiveness than PATHCRAWLER's native precondition. Our experiments also show that the late-precondition approach offers comparable performances to the native precondition and better performances than another alternative approach.

The paper is organized as follows. First, Sec. 2 gives a brief overview of precondition handling. Next, Sec. 3 describes the new method, while Sec. 4 evaluates it experimentally. Finally, Sec. 5 concludes the paper.

2 Related Work

Some test input generation tools allow to express the precondition in the tested language. Tools like Korat [2] are specifically designed to generate valid inputs based on such a precondition without considering the code of the program under test. However, a few code-based test generation tools may also handle the precondition that way. For instance, Java PathFinder [3] (generalized symbolic execution) and CUTE [4] (DSE) allow the user to provide a consistency check as a function in the tested language. The function is first solved, using the normal process of the tool. Similarly, Pex [5], a DSE tool for the .NET platform, treats Code Contracts, an embedded form of specification that is automatically translated into dynamic checks during compilation. For the precondition, assumption statements are placed **before** the code to be tested and handled as any other part of the code. Many DSE tools do not address the precondition problem specifically. However, at the cost of extra test cases, the precondition can be written as a conditional statement around the program code, leading to a solution almost equivalent to previous approaches. Like these tools, our approach proposes to encode the precondition in the tested language. However, it separates and delays the exploration of the precondition in order to minimize the exploration of the program code. To the best of our knowledge, this approach has never been used in other tools.

Another way to enforce the precondition is to describe how to construct a valid input rather than how to check whether a test case is valid. This method, sometimes called *finitization* [2], is dual to a classic precondition. Indeed, some complex structures are simpler to check than to construct, while others are better handled constructively.

The late-precondition method combines multiple symbolic executions. A related combination of symbolic executions is *compositional symbolic execution* proposed by Godefroid [6]. It aims at separating the symbolic execution of called functions in order to maximize reuse and to limit path exploration by generating so-called *function summaries*. Our late-precondition method has another objective. Indeed, it only separates the symbolic execution of two components (program and precondition) and does not create summaries. However, our late-precondition has a remarkable trait: the precondition is considered after the program. This ensures that the program's paths are explored only once without requiring to compute and store function summaries.

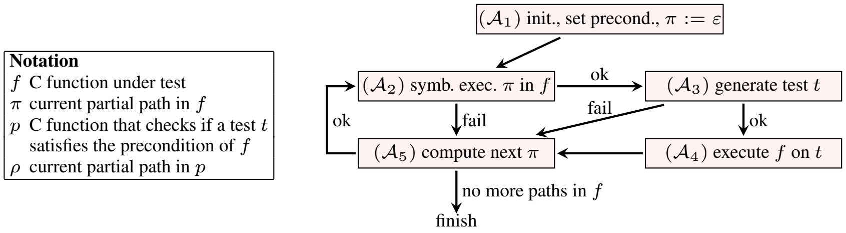
Notation
f C function under test
π current partial path in f
p C function that checks if a test t satisfies the precondition of f
ρ current partial path in p
($\mathcal{A}_1$) init., set precond., $\pi := \varepsilon$
($\mathcal{A}_2$) symb. exec. π in f
ok
($\mathcal{A}_3$) generate test t
fail
fail
ok
ok
($\mathcal{A}_5$) compute next π
($\mathcal{A}_4$) execute f on t
no more paths in f
finish

Fig. 1. Basic PathCrawler test generation method (with a native precondition)

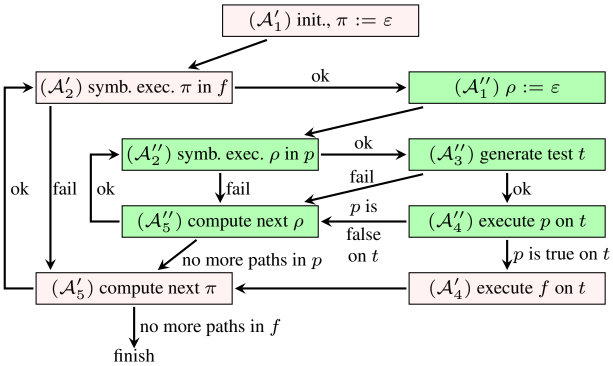
($\mathcal{A}'_1$) init., $\pi := \varepsilon$
($\mathcal{A}'_2$) symb. exec. π in f
ok
($\mathcal{A}''_1$) $\rho := \varepsilon$
($\mathcal{A}''_2$) symb. exec. ρ in p
ok
($\mathcal{A}''_3$) generate test t
ok
fail
ok
fail
fail
ok
($\mathcal{A}''_5$) compute next ρ
p is false on t
($\mathcal{A}''_4$) execute p on t
no more paths in p
p is true on t
($\mathcal{A}'_5$) compute next π
($\mathcal{A}'_4$) execute f on t
no more paths in f
finish

Fig. 2. Our late-precondition method using a precondition defined in a separate C function

3 Late-Precondition Method

Usual test generation in PathCrawler In Fig. 1 we briefly present (following [7, Sec.2.1]) the DSE-based test generation method for a C function f implemented in PathCrawler. Step $\mathcal{A}_1$ creates a logical variable for each input of f and posts the constraints for the precondition of f (given in an internal format). The depth-first exploration of program paths (steps $\mathcal{A}_2$-$\mathcal{A}_5$) starts with the empty path ε. $\mathcal{A}_2$ symbolically executes the current partial path π in f and posts corresponding path constraints, solved at $\mathcal{A}_3$ in order to generate a test t activating a path starting with π. If $\mathcal{A}_2$ or $\mathcal{A}_3$ fails, i.e., π is infeasible, then $\mathcal{A}_5$ continues directly to the next partial path in a depth-first search. If a test t is found, $\mathcal{A}_4$ executes f on t and observes the complete executed path and its results. Note that some solvers (e.g., Colibri the constraint solver used in PathCrawler) support incremental constraint solving. That is why, if the constraints are sent to the solver during the symbolic execution and if the solver detects the infeasibility of a path at steps $\mathcal{A}_2$-$\mathcal{A}_3$, the process skips $\mathcal{A}_4$ and goes to $\mathcal{A}_5$.

Late-precondition process Let us assume given the precondition of f defined by a C function p, returning true (a non-zero value) when the inputs are admissible for f. One could suggest to filter inputs by p before exploring f. It can be done when the precondition is a conjunction of elementary conditions (it is the case in the native precondition format of PathCrawler). The difficulty of treating any C function p is that

N	Example	Key part of	Native		Early		Late		
		the precond.	Time	Paths	Time	Paths	Time	Paths	Precond. calls
1	Merge	$\forall i,\ t_i \leq t_{i+1}$	3m32s	8718	117m43s	73644	4m8s	8142	31415
2	TriangMatrix	$\forall i \geq j,\ M_{ij} = 0$	—		38.6s	4893	27.5s	4093	557
3	PermutOrder	$\forall i \neq j,\ p_i \neq p_j$	17.9s	5153	23s	5179	23.2s	6071	2273
4	PermutOrder	$\forall i, \exists j,\ p_j = i$	—		2m12s	14491	25s	6027	2266

Fig. 3. Late precondition compared to native and early precondition treatment in PathCrawler

p can have several paths that may lead to an accepting `return` statement since a C precondition may encode a complex logic formula with disjunctions. How to cover, without repetitions, every program path in f by a test executing an accepting path in p?

Fig. 2 presents our *late-precondition method*. It consists in "exploring p after f", that is, searching, for each partial path π of f, a test accepted by p **after** posting the path constraints of π at $\mathcal{A}'_2$. The steps $\mathcal{A}'_i$ explore the paths of f in the same manner as the classical DSE method, presented above, in Fig. 1, except that the test generation step $\mathcal{A}_3$ is replaced by another DSE-like exploration $\mathcal{A}''_1$-$\mathcal{A}''_5$ for the precondition p. At the steps $\mathcal{A}''_1$-$\mathcal{A}''_5$, the process keeps in the constraint store the constraints for π all the time and adds those for the current partial path ρ in p when necessary. If $\mathcal{A}''_3$ finds a test t satisfying the precondition, t also satisfies the path constraints of π and the exploration of p stops. Otherwise, the process explores all paths of p to check that no admissible test executes the partial path π of f.

This method treats a C precondition in a completely automatic way and is available online [8] (see e.g. example MergePrecond). It never considers again the same path of f. The "exploring p before f" approach cannot guarantee this property, so the same path in f may be covered several times. In addition, our technique allows us to continue to benefit from incremental constraint solving approach (where the constraints of the same partial path π are never re-posted and re-solved again), one of the main forces of the PathCrawler method.

4 Experimental Evaluation

Fig. 3 presents selected experiments of path test generation with PathCrawler for some typical examples and compares our technique with a native precondition (when it can be expressed so in PathCrawler) and with an early C precondition called before (or in the beginning of) the function under test. The indicators include total test generation time, the number of explored (covered and infeasible) paths and, for the late treatment, number of calls to the precondition function (step $\mathcal{A}''_4$ in Fig. 2). The third column illustrates the form of the *essential part* of the precondition (shown in italic below). Ex. 1 is a merge of two given *sorted arrays* t, t' into a third one. Ex. 2 checks for a given *upper triangular matrix* M if $M^2 = 0$. Ex. 3 and Ex. 4 contain the same function computing the order of a given *permutation* p, but this property is ensured in different ways: we check that $p : \{0, \dots, n-1\} \to \{0, \dots, n-1\}$ is *injective* in Ex. 3 and *surjective* in Ex. 4. For simpler examples of preconditions (over a few variables without quantifiers), no noticeable difference in performances has been recorded.

Late precondition method appears to be more expressive than native precondition and has a limited overhead. It is more convenient for C developers to write precondition directly in C, using existing code fragments and/or familiar notation. Unlike in the early precondition treatment, our late precondition technique does not explore each path in the function under test several times for each path in the precondition function, that avoids to uselessly consider again already covered paths.

It is natural to expect that this feature brings a valuable benefit for preconditions with disjunctions or existential quantifications (like in Ex. 4) since the precondition likely has multiple accepting paths for a given path in the program under test. Interestingly, even for programs with only conjunctive and universally quantified preconditions, our technique may significantly reduce the global number of paths to be explored (cf. Ex. 1). We also observe that the late precondition treatment appears to be less sensitive to the form of the precondition: as illustrated by Ex. 3–4, writing logically the same precondition in a different way can result in a significant loss of performances for the early precondition while the late precondition treatment is not affected so much. Finally, we notice that a potentially great number of calls to (an efficient executable version of) the precondition does not dramatically slow down test generation (cf. Ex. 1–4).

5 Conclusion

We propose a late-precondition method for dynamic symbolic execution that combines at least two benefits. First, it takes as input an executable precondition written in the tested language, i.e., C for PATHCRAWLER. Such a precondition is easier to write for developers and can be very expressive. Second, the method ensures that paths of the function under test are considered once and only once during test generation. This notably allows high path coverage, where each uncovered path is either infeasible or outside chosen limits (e.g., on the number of loop iterations). This article also gives an initial experimental evaluation of the method. It was made possible through the implementation of the method in PATHCRAWLER. Our experiments report a little overhead with respect to a native precondition and significantly better performances with respect to an early precondition.

Moreover, C precondition appears particularly useful when combining static and dynamic analysis, notably in the SANTE tool [9] and in treating E-ACSL, an executable specification language for C [10]. Indeed, when combining tools with very different views on the program, the program's own language is often the most suitable common language to express a precondition. That is why C precondition is used in those works to encode preconditions given in Pre/Post specifications.

Future work includes further experiments with late precondition and studying the form of precondition for which it is more or less efficient. In case of a precondition filtering out a lot of inputs, one may expect that late precondition could be more expensive than the traditional approach since it may lead to the exploration of many irrelevant program paths. Further improvements (such as a combination with a summarized early precondition, or early treatment of a conjunctive part of the precondition) could improve the performances in more cases.

References

1. Botella, B., Delahaye, M., Hong-Tuan-Ha, S., Kosmatov, N., Mouy, P., Roger, M., Williams, N.: Automating structural testing of C programs: Experience with PathCrawler. In: AST'09
2. Milicevic, A., Misailovic, S., Marinov, D., Khurshid, S.: Korat: A tool for generating structurally complex test inputs. In: ICSE'07
3. Visser, W., Păsăreanu, C.S., Khurshid, S.: Test input generation with Java PathFinder. In: ISSTA'04
4. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: ESEC/FSE'13
5. Barnett, M., Fahndrich, M., de Halleux, P., Logozzo, F., Tillmann, N.: Exploiting the synergy between automated-test-generation and programming-by-contract. In: ICSE'09
6. Godefroid, P.: Compositional dynamic test generation. In: POPL'07
7. Kosmatov, N.: All-paths test generation for programs with internal aliases. In: ISSRE'08
8. Kosmatov, N.: Online version of PathCrawler. (2010–2013) http://pathcrawler-online.com/.
9. Chebaro, O., Kosmatov, N., Giorgetti, A., Julliand, J.: Program slicing enhances a verification technique combining static and dynamic analysis. In: SAC'12
10. Delahaye, M., Kosmatov, N., Signoles, J.: Common specification language for static and dynamic analysis of C programs. In: SAC'13