A Late Treatment of C Precondition in Dynamic Symbolic Execution

Mickaël Delahaye
University Grenoble-Alpes, LIG
38041 Grenoble, France
E-mail: firstname.lastname@imag.fr

Nikolai Kosmatov
CEA, LIST, Software Safety Laboratory
PC 174, 91191 Gif-sur-Yvette, France
E-mail: firstname.lastname@cea.fr

***Abstract*—Relevance of automatically generated test cases depends on an appropriate definition of a test context, or precondition. This paper presents a novel method for handling a precondition in dynamic symbolic execution (DSE) testing tools. This method allows PathCrawler, a DSE tool for C programs, to accept a precondition defined as a C function. It provides a simple way to express a precondition even for developers who are not familiar with specification formalisms. It has also proven useful when combining static and dynamic analysis.**

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

I. Introduction

In software testing, the *precondition* (often called *test context*) specifies the domain of the inputs on which the program under test should be tested. This ability to select the test input domains is very important. Indeed, it allows concentrating the testing effort on some part of the programs' possible inputs. The most common use of the test precondition is to select only the inputs for which the behavior of the program is specified. In that 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.

However, in the case of automated test input generation tools, the *precondition* leads to two interesting challenges. First, because the input is selected automatically, one must encode the precondition into a formalism understood by the tool. Second, the tool must take into account the precondition in its 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 format specific to the tool. 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. First Sec. II gives a brief overview of precondition handling. Then Sec. III describes the new method. Finally Sec. IV concludes this paper.

This work has been partly funded by the French-government Single Inter-Ministry Fund (FUI) through the IO32 project (Instrumentation and Tools for 32-bit Microcontrollers).

II. Related Work

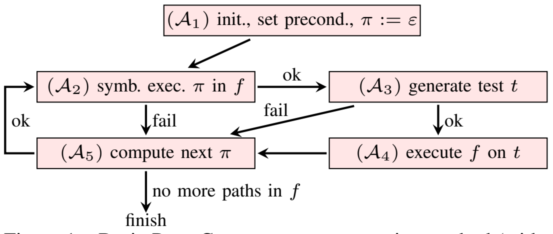
Some test input generation tools allow to express the precondition in the tested language. For instance, Java PathFinder [2] (generalized symbolic execution) and CUTE [3] (DSE) allow the user to provide a consistency check as a function in the tested language. The function is first solved, using the normal process of the tools. Similarly, Pex [4], a DSE tool for the .NET platform, treats Code Contracts, an embedded form of specification that automatically translates 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.

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* [5], is complementary to classic precondition. Indeed, some complex structures are simpler to check than to construct, while others are better handled constructively.

III. Late-Precondition Method

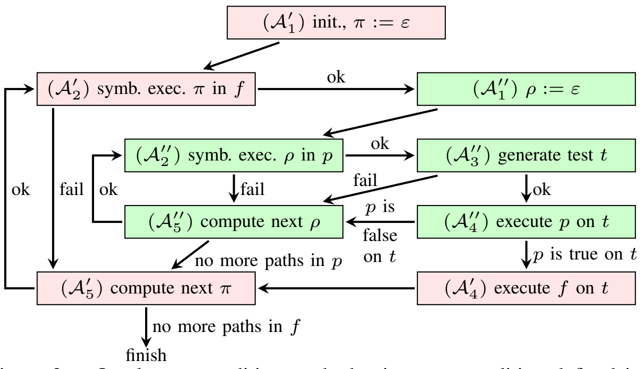
Usual test generation in PathCrawler*:* In Fig. 1 we briefly present (following [6, Sec.2.1]) the DSE-based test generation method for a C function f implemented in PathCrawler. Step $\mathcal{A}_1$ creates a logical variable for each input of f and posts the constraints for the precondition of f (given 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

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

Figure 1. Basic PATHCRAWLER test generation method (with a precondition defined in an internal format)



$(\mathcal{A}'_1)$ 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
p is false on t
$(\mathcal{A}''_5)$ compute next ρ
$(\mathcal{A}''_4)$ 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

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

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 internal precondition format of PATHCRAWLER). The difficulty of treating any C function p is that 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 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 [7] (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.

IV. 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 the test. This notably allows high path coverage, where each uncovered paths is either infeasible or outside chosen limits (e.g., on the number of loop iterations). This method was implemented in the tool PATHCRAWLER. It also appears very useful when combining static and dynamic analysis, notably in the SANTE tool [8] and in treating E-ACSL, an executable specification language for C [9]. Indeed, when combining tools with very different views on the program, the program's own language is often the only suitable common language to express a precondition. That is why C preconditions have been used in those works to encode preconditions given in Pre/Post specifications.

REFERENCES

[1] B. Botella, M. Delahaye, S. Hong-Tuan-Ha, N. Kosmatov, P. Mouy, M. Roger, and N. Williams, "Automating structural testing of C programs: Experience with PathCrawler," in *AST'09*.

[2] W. Visser, C. S. Păsăreanu, and S. Khurshid, "Test input generation with Java PathFinder," in *ISSTA'04*.

[3] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *ESEC/FSE-13*, 2005.

[4] M. Barnett, M. Fahndrich, P. de Halleux, F. Logozzo, and N. Tillmann, "Exploiting the synergy between automated-test-generation and programming-by-contract," in *ICSE'09*.

[5] A. Milicevic, S. Misailovic, D. Marinov, and S. Khurshid, "Korat: A tool for generating structurally complex test inputs," in *ICSE'07*.

[6] N. Kosmatov, "All-paths test generation for programs with internal aliases," in *ISSRE'08*.

[7] ——, "Online version of PathCrawler." 2010–2013, http://pathcrawler-online.com/.

[8] O. Chebaro, N. Kosmatov, A. Giorgetti, and J. Julliand, "Program slicing enhances a verification technique combining static and dynamic analysis," in *SAC'12*.

[9] M. Delahaye, N. Kosmatov, and J. Signoles, "Common specification language for static and dynamic analysis of C programs," in *SAC'13*, to appear.