# Enhancing Symbolic Execution for Coverage-Oriented Testing ⋆

Sébastien Bardin Nikolai Kosmatov Mickaël Delahaye
CEA, LIST, Laboratoire pour la Sûreté du Logiciel
91191, Gif-sur-Yvettes, France
first.name@cea.fr

***Abstract*—Automatic (code-based) test data generation is a major topic in software engineering, and Dynamic Symbolic Execution (DSE) is a very promising approach to this problem. However, while DSE inherently covers feasible paths of the program under test, practical testing is more concerned with fulfilling so called coverage criteria, such as instruction coverage, branch coverage, MCDC (in aeronautic) or mutations. Three problems arise for DSE in this context. First, path coverage may not be adapted to the criteria under consideration. Second, some of the coverage requirements may be infeasible. Third, we need to be able to handle a large range of different coverage criteria. We propose three ingredients to tame these issues: a unified management of a large class of coverage criteria through labels (i.e. reachability objectives), a variant of DSE designed to handle explicit coverage requirements at only a reasonable cost, and a combination of well-known static analyses for detecting infeasible coverage requirements. These results have been implemented into the LTest plugin of the open-source software analyzer Frama-C. We also present new results, including experiments on a weak form of the MCDC criterion and a combination of DSE with infeasibility detection.**

***Keywords*—Testing, symbolic execution, coverage criteria**

## I. Introduction

Automatic (code-based) test data generation is a major topic in software engineering, and Dynamic Symbolic Execution (DSE) [11], [14] is a very promising approach to this problem. While an old idea [15], Symbolic Execution has known a regain of interest in the mid 2000's [13], [17], [19], leading to many academic tools and case-studies [4], [7], [8], [18]. However, while DSE inherently covers feasible paths of the programs under test, practical testing is more concerned with fulfilling so called *coverage criteria* [1], [20], such as instruction coverage, branch coverage, **MCDC** [10] or mutations [12]. Three problems arise for DSE in this context:

- path coverage may not be adapted to the criterion under consideration, e.g. path coverage does not ensure multiple-condition coverage;
- since coverage requirements are syntactically defined from the program under test (without considering its semantic), some or even many coverage requirements may be infeasible, leading to a waste of efforts trying to cover these objectives, as well as to artificially low coverage ratios;
- finally, there exist many different classes of coverage requirements, and any coverage-oriented testing approach must be able to handle a large part of them.

We propose three solutions to these problems. First, we propose *labels* (reachability objectives) as a way of modeling many existing coverage criteria [6]. Second, we define a variant of DSE [6] which handles explicit coverage requirements at only a reasonable cost (i.e., a polynomial growth of the search space, while a standard approach induces an exponential blowup). Third, we use a combination of static analyses [3] in order to detect infeasible coverage requirements. These results have been implemented in the LTest plugin [2] of the open-source software analyzer Frama-C [9].

After reviewing this label-based testing framework, we present new experimental results on weak forms of the MCDC criterion and additional optimizations of the approach.

## II. Overview

**Background: symbolic execution.** Symbolic execution is considered as a very fruitful and promising approach to automatic test generation from source-code. Basically, the technique amounts to iterate over (a finite subset of) the paths of the program under test, and for each path to compute a so-called *path predicate*, i.e. a formula such that any input satisfying it is ensured to exercise the given path at runtime. The formula is then fed to an automatic solver (typically: SMT solver) to derive a new test input. A generic view of the algorithm is depicted in Algorithm 1.

**Algorithm 1:** Symbolic Execution algorithm

```
Input: a program $P$ with finite set of paths $Paths(P)$
Output: $TS$, a set of pairs $(t, \sigma)$, with $t$ a test input
        and $\sigma$ a path, such that $P(t)$ covers $\sigma$
$TS := \emptyset$;
$S_{paths} := Paths(P)$;
while $S_{paths} \neq \emptyset$ do
    choose $\sigma \in S_{paths}$; $S_{paths} := S_{paths} \setminus \{\sigma\}$;
    compute path predicate $\phi_\sigma$ for $\sigma$ ;
    switch solve($\phi_\sigma$) do
        case sat(t): $TS := TS \cup \{(t, \sigma)\}$;
        case unsat: skip;
    endsw
end
return $TS$;
```

⋆ Work partially funded by EU FP7 (project STANCE, grant 317753) and French ANR (project BINSEC, grant ANR-12-INSE-0002, and SOPRANO, grant ANR-14-CE28-0020).

**Labels.** Labels [6] are basically reachability objectives inserted into the program under test. They can perfectly emulate many standard criteria [6], ranging from basic ones (Instructions, Decisions, Conditions) to more advanced ones (e.g. side-effect free weak mutations). Labels are interesting here for two essential reasons: (1) they allow to manage in a unified way many different criteria, (2) they allow to reuse the whole machinery of program verification for dealing with coverage criteria issues, since reachability is at the heart of program verification. An encoding of a standard criterion is given in Figure 1. Readers can refer to [6] for more examples.

```
statement_1;
if (x==y && a<b)
  {...};
statement_3;
```

$\longrightarrow$

```
statement_1;
// l1: x==y
// l2: !(x==y)
// l3: a<b
// l4: !(a<b)
if (x==y && a<b)
  {...};
statement_3;
```

Condition (CC)

Fig. 1. Simulating the Condition Coverage criterion with labels

**The** LTEST **tool.** LTEST is a label-based white-box testing framework [2], implemented on top of Frama-C [9] and PathCrawler [19], which provides three main services: (1) computation of the coverage score of a given test suite, (2) automatic generation (resp. completion) of a test suite (resp. of a given test suite), (3) automatic detection of infeasible coverage requirements. And, thanks to labels, a wide range of coverage criteria are supported. A schematic overview of the platform is depicted in Figure 2.

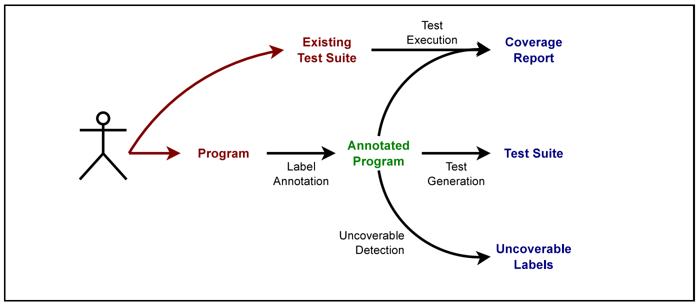

**Fig. 2:** User view of the LTEST platform

## III. Symbolic Execution and Coverage

**Label-oriented symbolic execution.** As already stated, path coverage does not entail certain other coverage criteria, especially it does not ensure label coverage. There is a straightforward way for taking labels into account in symbolic execution: each label can be hard-coded in the program under test with extra branching conditions [16]. Yet, this approach (denoted DSE') can yield a dramatic blowup of the search space [6]. We propose DSE$^\star$, a variant of DSE which takes advantage of *tight instrumentation* (cf. Figure 3) and *iterative label deletion* in order to manage labels in a very efficient way. Especially, it can be shown that the search space of DSE$^\star$ increases only polynomially in the number of labels w.r.t. the search space of DSE.

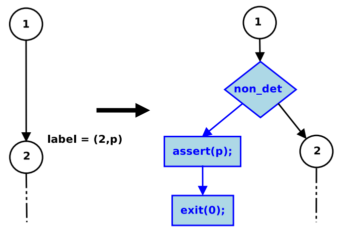

Fig. 3. Tight instrumentation

Experiments reported in [6] (summarized here in Table I), over 3 coverage criteria (**CC**, **MCC**, **WM**), clearly demonstrate the advantage of DSE$^\star$ w.r.t. DSE with straightforward instrumentation (DSE'). Moreover, the overall overhead w.r.t. standard DSE is kept low.

TABLE I. Overhead (slow-down) with respect to DSE

| | DSE' | DSE$^\star$ |
|---|---|---|
| Min | ×1.02 | ×0.49 |
| Median | ×1.79 | ×1.37 |
| Max | ×122.50 | ×7.15 |
| Mean | ×20.29 | ×2.15 |
| Timeouts$^*$ | 5 | 0 |

$^*$Overhead take into account timeouts, counted as 5400s (90min)

Finally, new experimental results reported in Table II show that, in terms of coverage, DSE$^\star$ does improve over DSE (which performs already well) and over random testing.

TABLE II. Label coverage ratios

| | Random | DSE | DSE' | DSE$^\star$ |
|---|---|---|---|---|
| Min | 37% | 61% | 62% | 62% |
| Median | 63% | 90% | 92% | 95% |
| Max | 100% | 100% | 100% | 100% |
| Mean | 70% | 87% | 88% | 90% |

Timeouts are excluded from the coverage computation.

**Infeasibility detection.** In order to detect infeasible labels, we reuse techniques from software verification. Indeed, a label is infeasible iff its negation is a valid assertion of the program under test, which is what software verification is about. We consider two different popular approaches, value analysis by abstract interpretation (VA) and weakest precondition calculus (WP), as well as a lightweight combination of both, denoted VA$\oplus$WP [3]. Results are reported in Table III (taken from [3]). The combination allows to detect almost all infeasible labels (98%), and performs significantly better than each approach taken in isolation.

**Combining DSE$^\star$ and VA$\oplus$WP.** Symbolic execution can take advantage of the detection of infeasible coverage requirements in two ways. First, the infeasibility information can be used to report more accurate coverage ratios (Table IV). Second, infeasible labels can be ignored by symbolic execution, and the search stops once all labels are covered (Table V).

TABLE III. DETECTION OF INFEASIBLE LABELS

| | **#Lab** | **#Inf** | **VA** | | **WP** | | **VA ⊕ WP** | |
|---|---|---|---|---|---|---|---|---|
| | | | **#d** | **%d** | **#d** | **%d** | **#d** | **%d** |
| Total | 1,270 | 121 | 84 | 69% | 73 | **60%** | 118 | **98%** |
| Min | | 0 | 0 | 0% | 0 | 0% | 2 | **67%** |
| Max | | 29 | 29 | 100% | 15 | 100% | 29 | 100% |
| Mean | | 4.7 | 3.2 | 63% | 2.8 | **82%** | 4.5 | **95%** |

#Lab: number of labels

#Inf: number of infeasible labels (manual inspection)

#d: number of detected infeasible labels

%d: ratio of detected infeasible labels

TABLE IV. CORRECTED COVERAGE RATIOS

| | Random | DSE | DSE$^\star$ | | |
|---|---|---|---|---|---|
| | | | norm. | VA | VA ⊕WP |
| Total | 67% | 81% | 90% | 96% | 99% |
| Min. | 37% | 61% | 62% | 80% | 91% |
| Med. | 63% | 90% | 95% | 100% | 100% |
| Max. | 100% | 100% | 100% | 100% | 100% |
| Mean | 70% | 87% | 90% | 96% | 99% |

For DSE$^\star$ VA⊕WP, ratios take into account the detected infeasible labels.
Timeouts are excluded from the coverage computation.

TABLE V. IMPROVING DSE$^\star$ EFFICIENCY

| | DSE$^\star$-OPT vs DSE$^\star$ |
|---|---|
| Min. | 0.96× |
| Med. | 1.46× |
| Max. | 592.54× |
| Mean | 49.04× |

performance speedup is relative to DSE$^\star$

**The GACC coverage criterion.** We consider now a more demanding coverage criterion, namely **GACC** [1], a weak interpretation of **MCDC** (a.k.a. shortcut MCDC). The criterion is indeed demanding, but also industrially relevant (critical systems) and expressible by labels [16] (while full **MCDC** is not). Experiments (reported in Tables VI and VII) confirm that **GACC** is a challenging criterion for automatic tools. Yet, while the overhead of DSE$^\star$ is more important than for other criteria, it is still affordable in most cases, and the technique achieves very good coverage — much better than random testing or standard DSE. Additional optimizations allow to keep overhead low.

TABLE VI. OVERHEADS (SLOW-DOWN) W.R.T. DSE FOR GACC

| | DSE' | DSE$^\star$ | |
|---|---|---|---|
| | | norm. | OPT |
| Min | 1.44× | 1.41× | 1.38× |
| Med | 3.76× | 1.81× | 1.44× |
| Max | 130.79× | 59.40× | 3.14× |
| Mean | 21.99× | 10.55× | 1.85× |
| Timeouts* | 1 | 0 | 0 |

* For DSE', a timeout counts as a 5400s (90min).

## IV. CONCLUSION

Symbolic execution is a promising approach for automatic code-based testing. Yet, while the primary technique is essentially based on path exploration, code-based testing is often about coverage criteria, leading to a sort of mismatch between what a symbolic execution engine does, and what it should do.

TABLE VII. LABEL COVERAGE RATIOS FOR **GACC**

| | Random | DSE | DSE$^\star$ | |
|---|---|---|---|---|
| | | | norm. | OPT |
| Min | 47% | 62% | 64% | 72% |
| Med | 55% | 76% | 88% | 96% |
| Max | 100% | 100% | 100% | 100% |
| Mean | 60% | 78% | 85% | 91% |

For DSE$^\star$ OPT-1-2, ratios take into account the detected infeasible labels.
Timeouts are excluded from the coverage computation.

We have proposed different techniques to enhance symbolic execution in order to overcome these problems. Our framework is based on three ingredients: (1) a "low-level" coverage criterion (labels), able to encode a large variety of standard coverage criteria; (2) a dedicated variant of DSE, optimized for label coverage; (3) the use of state-of-the-art verification technologies in order to automatically detect infeasible test requirements.

Experiments show the potential of the approach. Especially, symbolic execution can take a clear advantage of static analysis in terms of efficiency and reported coverage ratios, and the method still obtains excellent results on very demanding coverage criteria, such as weak mutations or **GACC**. Moreover, there is still room for improvements, for example through combining labels with search-based optimizations of symbolic execution, such as [5].

## REFERENCES

[1] P. Ammann, A. J. Offutt: Introduction to software testing. Cambridge University Press, New York (2008)

[2] S. Bardin, O. Chebaro, M. Delahaye, N. Kosmatov: An All-in-One Toolkit for Automated White-Box Testing. In: TAP 2014. Springer, Heidelberg (2014)

[3] S. Bardin, M. Delahaye, R. David, N. Kosmatov, M. Papadakis, Y. Le Traon, J.-Y. Marion: Sound and quasi-Complete Detection of Infeasible Test Requirements. In: ICST 2015. IEEE, Los Alamitos (2015)

[4] S. Bardin and P. Herrmann. Structural Testing of Executables. In: IEEE ICST 2008. IEEE, Los Alamitos (2008)

[5] S. Bardin and P. Herrmann. Pruning the search space in path-based test generation. In: ICST 2009. IEEE, Los Alamitos (2009)

[6] S. Bardin, N. Kosmatov, F. Cheynier.: Efficient Leveraging of Symbolic Execution to Advanced Coverage Criteria. In: ICST 2014. IEEE, Los Alamitos (2014)

[7] C. Cadar, D. Dunbar, D. Engler: KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In: OSDI 2008. Usenix Association (2008)

[8] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill and D. R. Engler. EXE: automatically generating inputs of death. In: CCS 2006. ACM

[9] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, B. Yakobowski: Frama-C - A Software Analysis Perspective. In: SEFM 2012. Springer (2012)

[10] Chilenski, J. J., Miller, S. P.: Applicability of modified condition/decision coverage to software testing. Software Engineering Journal, 9(5), 193–200 (1994)

[11] C. Cadar, K. Sen: Symbolic execution for software testing: three decades later. Commun. ACM, 56(2), 2013.

[12] R. A. DeMillo, R. J. Lipton, A. J. Perlis: Hints on test data selection: Help for the Practicing Programmer. Computer, 11(4), 34–41

[13] P. Godefroid, N. Klarlund and K. Sen. DART: Directed Automated Random Testing. In: PLDI 2005. ACM

[14] P. Godefroid, M. Y. Levin, D. A. Molnar: SAGE: whitebox fuzzing for security testing. Commun. ACM 55(3): 40–44 (2012)

[15] J. C. King. Symbolic execution and program testing. Communications of the ACM, 19(7), july 1976.

[16] R. Pandita, T. Xie, N. Tillmann and J. de Halleux. Guided test generation for coverage criteria. In: ICSM 2010. IEEE

[17] K. Sen, D. Marinov, G. Agha: CUTE: A Concolic Unit Testing Engine for C. In: ESEC/FSE 2005. ACM

[18] N. Tillmann and J. de Halleux. Pex-White Box Test Generation for .NET. In: TAP 2008. Springer

[19] N. Williams, B. Marre and P. Mouy. On-the-Fly Generation of K-Path Tests for C Functions. In: ASE 2004. IEEE (2004)

[20] H. Zhu, P. A. V. Hall and J. H. R. May. Software Unit Test Coverage and Adequacy. In: ACM Computing Surveys, vol. 29(4), 1997