Science of Computer Programming 207 (2021) 102641

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

# Specify and measure, cover and reveal: A unified framework for automated test generation

Sébastien Bardin [a], Nikolai Kosmatov [a,b,*], Michaël Marcozzi [a,c], Mickaël Delahaye [d]

[a] Université Paris-Saclay, CEA, List, Palaiseau, France
[b] Thales Research & Technology, Palaiseau, France
[c] Imperial College London, United Kingdom
[d] DGA, Bruz, France

ARTICLE INFO

*Article history:*
Received 14 April 2020
Received in revised form 5 March 2021
Accepted 8 March 2021
Available online 17 March 2021

*Keywords:*
Software testing
Test generation
Symbolic execution
Static analysis
Coverage criteria

ABSTRACT

Automatic test input generation (ATG) is a major topic in software engineering, analysis and security. In this paper, we bridge the gap between state-of-the-art white-box ATG techniques, especially Dynamic Symbolic Execution, and the diversity of test objectives that they may be used to cover in practice, including many of those defined by common source-code coverage criteria. We define a new coverage specification mechanism, called labels, for *specifying* test objectives, and prove it to be both expressive and amenable to efficient automation. We present an efficient approach for detecting – *revealing* – infeasible (i.e. uncoverable) test objectives expressed as labels. We demonstrate that *measuring* the achieved coverage can be efficiently performed for labels. Finally, we propose an innovative extension of DSE resulting in an efficient support for label *coverage*, while the existing naive approach induces an exponential blow-up of the search space. Experiments show that our ATG technique yields very significant savings and confirm the interest of infeasible label detection, enabling to lift DSE to label coverage with only a slight overhead. Overall, we show that label coverage provides the basis of a rich framework allowing one to express and handle test objectives from various contexts in an efficient and generic manner. To illustrate this framework, we describe LTest, an all-in-one testing toolset based on labels and used in the industry, which offers automatic program annotation, ATG, coverage measurement and detection of infeasible test objectives.

## 1. Introduction

**Context.** Efficient test input generation is a major issue for software engineering, analysis and security, so that a tremendous amount of work has been carried out to develop *Automatic Test Generation* (ATG) techniques and apply them in various contexts. These techniques[1] may select tests randomly or craft them to cover specific targets in the code, relying for example

* Corresponding author at: Thales Research & Technology, Palaiseau, France.
*E-mail addresses:* sebastien.bardin@cea.fr (S. Bardin), nikolaikosmatov@gmail.com (N. Kosmatov), michael.marcozzi@cea.fr (M. Marcozzi), mickael.delahaye@inria.fr (M. Delahaye).

[1] The scope of this paper does not include model-based testing techniques, aiming at producing inputs covering the specification of the code under test. We consider here only deterministic sequential programs, even if extending the presented techniques to concurrent programs can be an interesting work perspective.

https://doi.org/10.1016/j.scico.2021.102641
0167-6423/© 2021 Elsevier B.V. All rights reserved.

on search meta-heuristics or symbolic analysis to produce the relevant tests. In the latter case, advances in constraint solving and dynamic analysis have led to the surge of *Dynamic Symbolic Execution* (DSE) [45,68,73], implemented into many tools (e.g. [30,31,37,38,46,70]) leading to impressive case-studies (e.g. [37,38,48]).

In so-called white-box testing, ATG tools are used to fulfill various kinds of *test objectives* defined from the code of the program under test. For instance, they can be employed to search for inputs triggering specific types of failures during code execution (like buffer overflows) or capable of stressing specific zones in the code (like those affected by a software patch). ATG may also be used to generate a general test suite for a piece of software, which is then passed to one or several external oracles, in order to assess for example functional correctness, security or performance. The more different code behaviors are exercised by the test suite, the better. A standard way of measuring this diversity involves coverage criteria [25,75] (a.k.a. adequacy criteria). We focus in this paper on *source code* coverage criteria, simply referred to as *coverage criteria* in the following. Many such criteria have been defined along the years, from basic control-flow or data-flow criteria to mutations [43] and MCDC [42].

**Problem.** Common white-box ATG techniques may face issues to cope efficiently with the diversity of test objectives that they are confronted with in practice. For example, DSE mostly follows an exhaustive exploration of the path space of the program under test, aiming typically at covering most execution paths up to a given bound. While such a path-oriented exploration proves successful in some contexts, it is well known that the resulting test suite can miss interesting behaviors related to data rather than control. Moreover, standard DSE does not support coverage objectives defined over artifacts not explicitly present in the code, such as multiple-condition coverage [25] or mutations, while it could efficiently guide test generation towards covering them.

Another important issue is that many white-box test objectives are defined in a *structural* way, i.e. expressed in terms of generic code artifacts (e.g. cover all instructions, all decisions, all conditions, etc.), without taking into account the semantics of the program. This leads to the situation when some of the resulting concrete test objectives (instructions, decisions, conditions, etc.) can be impossible to activate by a test case – they are *infeasible*, i.e. uncoverable. Infeasible test objectives waste the test generation effort and prevent testers from measuring the objectives coverage ratio (proxy for test effectiveness [2,3]) precisely.

**Goals.** Our first objective is to adapt a state-of-the-art ATG technique, namely DSE, to make it able to cover efficiently a wide class of test objectives derived from the source code of the tested program. Recent works have aimed at lifting DSE to various coverage criteria [50,64–67,76], or improving DSE bug-detection abilities by making explicit run-time error conditions [40,47,52]. These approaches are mainly based on an instrumentation of the code under test and allow for black-box reuse of existing DSE tools. However, they come at a high price since they may induce a blow-up of the path space and a significant overhead (previous work [50, Table 2] reports on a 272x average time-overhead, with a worst case of 2,000x). While still relying on instrumentation of the tested program and emphasizing black-box reuse of DSE tools as much as possible, our goal is to provide DSE support for even more kinds of objectives with a minimal and acceptable overhead.

Our second objective is to automate three additional key testing services, which we argue should be typically performed before any actual input generation. First, we offer a simple, generic and formal way to *specify* what the test objectives are, for many kinds of programs and testing contexts. Second, we enable one to efficiently detect – *reveal* – those of the test objectives that are actually infeasible (or at least a significant part of them). Third, we provide means to *measure* the coverage ratio of any existing test suite w.r.t. the feasible specified objectives. One can then leverage our adapted DSE to *cover* efficiently those of the feasible specified objectives that have not been covered yet.

**Approach.** We introduce *labels*, i.e. predicates attached to given program instructions. Labels were named with reference to C labels, which attach an identifier to a given program instruction (while labels attach predicates instead). We define *label coverage*, a new source code coverage criterion which appears to be both expressive and amenable to efficient automation. A label is covered if a test execution reaches the instruction and satisfies the predicate. Actually, labels can be thought of as a *convenient specification mechanism for test objectives*, enabling to simulate notably many common classes of coverage criteria in a unified way. This idea encompasses and extends several existing works [40,47,50,52,66,76].

We propose two novel ways of taming the blow-up that may appear while trying to cover labels with DSE. Namely, we introduce a *tight instrumentation*, where “tight” is made precise in the paper, and a coupling between DSE and label coverage named *iterative label deletion*. Their combination results in a much more effective support for label coverage in DSE. In addition, both techniques can be implemented using black-box DSE tools. We also show that labels are amenable to an efficient coverage measurement and an efficient detection of infeasible test objectives. Experiments reveal that DSE with tight instrumentation, iterative label deletion and infeasible label detection reaches better label coverage ratios than vanilla DSE, with only a slight time overhead.

Overall, we demonstrate that label coverage provides the basis of a rich framework allowing one to express and handle various test objectives in an efficient and generic manner.

**Contributions.** The purpose of the paper is to provide a complete panorama of various efforts related to label coverage, including basic definitions, key results and techniques, as well as a summary of most recent extensions and industrial applications. Our main contributions are the following:

- We show that label coverage is expressive enough to faithfully emulate notably many standard white-box coverage criteria, from decision or condition coverage (Theorem 1) to advanced logic criteria (Theorem 2) and a substantial subset of weak mutations (the side-effect free fragment, Theorem 3). Labels can thus be seen as a convenient and powerful specification mechanism for coverage criteria.
- We demonstrate that infeasible labels can be detected by existing assertion checkers after translating labels into assertions (Lemma 4).
- We formally characterize the properties of the naive instrumentation used in previous works lifting DSE to some kinds of the test objectives that labels can encode. This instrumentation provides a sound way to achieve label coverage and leads to very efficient coverage score computation. However, it also yields an exponential increase as well as a *complexification* of the paths space (Theorem 7).
- We propose DSE⋆, a variant of DSE with efficient handling of all the kinds of test objectives encodable by labels. This approach relies on *tight instrumentation* and *iterative label deletion* to reduce the complexity introduced by labels. Tight instrumentation yields only a linear growth of the paths space without any complexification (Theorem 10). Both techniques are orthogonal and allow for a significant speed-up. Moreover, they can be both implemented either within the DSE algorithm or using existing DSE tools in a black-box manner.
- We have implemented DSE⋆ inside the PathCrawler DSE tool [73,28]. Experiments show that tight instrumentation and iterative label deletion yield very significant reductions of both the search space and computation time compared to naive instrumentation (several orders of magnitude speed-up in some cases).
- Finally, we describe LTest, an all-in-one ATG toolset based on label coverage. Along with DSE⋆-based test generation, LTest offers several integrated services: program annotation with labels, label coverage score computation, as well as detection of infeasible labels via static analysis. Our experiments with LTest demonstrate that better ATG for label coverage can be achieved at a very reasonable cost compared to vanilla DSE. For example, considering the test objectives from the advanced **MCDC** criterion over our benchmark programs, DSE⋆ with infeasible label detection has a mean 1.85x time overhead compared to vanilla DSE, while the mean reported coverage ratio is increased from 78% to 91%.

**Outline.** First, we present a motivating example in Section 2. After detailing our basic notation (Section 3), we define labels and explore their expressiveness (Section 4). Next, we focus on automation. Detection of infeasible labels is described in Section 5. The naive instrumentation is studied in Section 6.1 and the optimized DSE⋆ approach is presented in Section 6.2. Thereon, we describe LTest, the automated testing framework based on labels (Section 7). Our experiments are presented in Section 8, followed by the discussion of threats to their validity in Section 9. Recent extensions and applications of labels are summarized in Section 10. Finally, we discuss related work (Section 11) and provide a conclusion (Section 12).

**Earlier works.** The present paper attempts to offer the first consolidated panorama of more than six years of research and industry transfer efforts around labels. As a consequence, this paper provides an integrated, as well as carefully revised, enhanced and extended version of several previously published works. More precisely, Sections 3.1, 3.3, 4, 6, 8.1, 11 are based on previous work presented at ICST 2014 [32]. The principles described in Section 5 were first introduced at ICST 2015 [27]. Section 7 was originally presented at TAP 2014 [26]. These earlier works have been extended in several ways.

Firstly, we have provided more explanations wherever possible, additional examples of criteria simulation, clearly stated theorems for all theoretical results, a thorough discussion on the limitations of labels, a better description of the experimental protocols, and an extended related work. A new motivating example (Section 2) has also been included to emphasize the integrated vision of the various aspects of testing addressed in the paper. We have also better presented all the coverage criteria considered in the paper (Section 3.2).

Secondly, the experiments detailed in Section 8 have been extended compared to those presented in the original papers. Section 8.1 has been strengthened with a new comparison with random testing and a better comparison with standard DSE (e.g. including coverage information). Section 8.2 was added to provide a novel comparison between vanilla DSE and DSE* with infeasible label detection. Section 8.3 was added to provide results involving an advanced coverage criterion (**MCDC**). A new section was also added to discuss threats to validity (Section 9).

Finally, all recent extensions [8,9,19,1], applications [35] and industrial adoption efforts [33,34] are now synthesized in Section 10.

## 2. Motivating example

In this paper, we introduce a unified framework for automated test input generation. Conceptually, we argue that the test generation process should be divided into four main activities: (1) *specify* the test objectives, (2) *reveal* the infeasible objectives, (3) *measure* the objective coverage rate of the existing tests and (4) *generate* new tests to cover the uncovered test objectives. We illustrate now how our framework handles these four activities in two of the most common use cases of test generation tools, namely crafting coverage-adequate test suites and detecting runtime failures. To do so, we consider the problem of automated test generation for the simple C function `numPos` in Fig. 1.

```
// Returns how many of the two inputs are strictly positive
int numPos(int a, int b) {
  int n = 0;
  if( a > 0 ) n++;
  if( b > 0 ) n++;
  return n;
}
```

**Fig. 1.** Function `numPos`.

```
int numPos(int a, int b) {
  int n = 0;
  // l1: a > 0; l2: a <= 0
  if( a > 0 ) n++;
  // l3: b > 0; l4: b <= 0
  if( b > 0 ) n++;
  return n;
}
```

**Fig. 2.** Function `numPos` with test objectives for Decision Coverage encoded as labels.

```
int numPos(int a, int b) {
  int n = 0;
  if( a > 0 ) {
    // l5: n == MAX_INT
    n++;
  };
  if( b > 0 ) {
    // l6: n == MAX_INT
    n++;
  };
  return n;
}
```

**Fig. 3.** Function `numPos` with test objectives for integer overflow detection encoded as labels.

### *2.1. Specifying test objectives*

To be efficient, automated test generation should be driven by precise and specific objectives to fulfill. Our framework introduces a generic test objective specification language called labels, which can encode many of the diverse kinds of test objectives occurring in the diverse use cases of test generation tools.

When a test generation tool is used to craft a test suite for a given program, the testers often aim at satisfying one of the many existing code coverage criteria, which define a set of syntactic elements in the program as the test objectives to be covered. Higher coverage test suites have indeed better chances to find bugs [2,3]. For a significant proportion of these criteria, labels can be used to easily specify the syntactic elements which should be covered in the program under test to satisfy the criterion. As a simple example, the Decision Coverage criterion requires that running the test suite should cover all the branches in the control-flow graph of the program. For the program in Fig. 1, this leads to four test objectives corresponding to the two branches of each of the two conditional statements. Each of these four test objectives can be encoded by one of the four labels `l1`, `l2`, `l3` and `l4` on Fig. 2. A label is basically a code assertion and covering the test objective encoded by a label means crafting a test that makes the program execution reach and fulfill the assertion. For example, one can check that covering `l1` is equivalent to making the condition of the first conditional statement true, i.e. to covering the then branch of this statement.

Test generation tools can also be used to probe a program for different kinds of runtime failures (see [4] for a successful example), like integer overflows. For the program in Fig. 1, this means crafting test inputs able to make one of the two `n++` statements overflow. These two test objectives can be encoded with the labels `l5` and `l6` on Fig. 3.

Given a program under test and a coverage criterion (or well-defined kind of runtime failures to probe for), our framework enables annotating automatically the program with the corresponding labels.

### *2.2. Revealing infeasible test objectives*

Many of the test objectives considered in automated test generation are purely syntactic and thus blind to the semantics of the program under test. As a consequence, a significant proportion of them can turn out to be infeasible, i.e. no input can lead to an execution satisfying them. For example, both labels `l5` and `l6` on Fig. 3 are infeasible, because `n` can only range between 0 and 2 during program execution.

```
int numPos(int a, int b) {
  int n = 0;
  if( a > 0 ) {
    assert(n != MAX_INT); // Never violated => l5 infeasible
    n++;
  };
  if( b > 0 ) {
    assert(n != MAX_INT); // Never violated => l6 infeasible
    n++;
  };
  return n;
}
```

**Fig. 4.** Assertions to detect infeasible test objectives for the test objectives of Fig. 3.

Infeasible objectives are a threat to the efficiency of test generation tools, because a significant part of the test budget might be lost trying to cover them. Our framework proposes to deal with this issue in a generic way, by introducing a sound approach to prune out infeasible objectives encoded as labels. In a nutshell, proving that a label is infeasible is equivalent to proving that its corresponding opposite assertion (i.e with the negated predicate) can never be violated during any program execution. Proving the latter is a standard feature of many formal verification or model-checking tools, to which this proof can be delegated. While label infeasibility is not decidable, i.e. the tool may not always be able to conclude, leaving the status of some labels unresolved, our experiments show that existing tools are able to flag many infeasible labels in practice. For example, for the program of Fig. 4, such a tool could show that none of its two assertions can be violated, hence we can deduce that `l5` and `l6` on Fig. 3 are infeasible.

### 2.3. Measuring the coverage

While building a test suite for a program, it is often useful to evaluate the strength of the produced test suite, to determine if additional tests should be generated to achieve a more acceptable coverage level or, on the contrary, if the test suite can be pruned to reduce the testing overhead. This is typically done by measuring which proportion of the test objectives from a chosen criterion are covered by the current test suite.

Our framework enables transparently instrumenting the program under test to measure the coverage level of an existing test suite, for any coverage criterion whose objectives can be encoded as labels. This can indeed be simply done through textually replacing any label by some code that reports the coverage of the corresponding objective. For example, considering the labels `l1` and `l2` of Fig. 2, the instrumented code could be:

```
...
int n = 0;
if (a > 0) report_covered_label("l1");
if (a <= 0) report_covered_label("l2");
if( a > 0 ) n++;
...
```

### 2.4. Generating test cases covering test objectives

While dozens of kinds of test objectives are used in the use cases of test generation tools, these different flavors of objectives are seen as dissimilar bases for automation, so that most tools only provide a direct support for a very small subset of them. Supporting new flavors of objectives is time-consuming. Our framework bridges the gap between the variety of test objective flavors and their limited support in test generation tools, by tailoring a state-of-the-art test generation approach, namely dynamic symbolic execution, to efficiently and generically cover objectives encoded as labels in the program under test. For example, considering the program of Fig. 2 and its four labels, our tailored dynamic symbolic execution would need only 4 trials before covering all the labels, while the common naive approach might necessitate up to 16 trials.

## 3. Background

### 3.1. Notation

Given a program $P$ over a vector $V$ of $m$ input variables taking values in a domain $D \triangleq D_1 \times \cdots \times D_m$, a test datum $t$ for $P$ is a valuation of $V$, i.e. $t \in D$. The execution of $P$ over $t$, denoted $P(t)$, is a path (or run) $\sigma \triangleq (loc_1, S_1) \ldots (loc_n, S_n)$, where the $loc_i$ denote control-locations (or simply locations) of $P$ and the $S_i$ denote the successive internal states of $P$ ($\approx$ valuation of all global and local variables as well as memory-allocated structures) before the execution of each $loc_i$. A test datum $t$ reaches a location *loc* with internal state $S$, denoted $t \leadsto_P (loc, S)$, if $P(t)$ is of the form $\sigma_1 \cdot (loc, S) \cdot \sigma_2$. A test suite *TS* is a finite set of test data.

Assume that for a given test objective $\mathbf{c}$, there is an adequate notion of covering (left unspecified for the moment), and we write $t \leadsto_P \mathbf{c}$ if test datum $t$ covers $\mathbf{c}$. We extend the notation for a test suite *TS* and a set of test objectives $\mathbf{C}$, writing $TS \leadsto_P \mathbf{C}$ when for any $\mathbf{c} \in \mathbf{C}$, there exists $t \in TS$ such that $t \leadsto_P \mathbf{c}$. A *(source-code based) coverage criterion* $\mathbb{C}$ is defined as a systematic way of deriving a set of test objectives $\mathbf{C} = \mathbb{C}(P)$ for any program under test $P$. A test suite *TS* satisfies (or achieves) a given coverage criterion $\mathbb{C}$ if *TS* covers $\mathbb{C}(P)$. When no confusion is possible, we can identify the coverage criterion $\mathbb{C}$ for a given program $P$ with the derived set of test objectives $\mathbf{C} = \mathbb{C}(P)$.

These definitions are generic and leave the exact definition of "covering" to the considered coverage criterion. For example, test objectives derived from the Decision Coverage criterion are of the form $\mathbf{c} \triangleq (loc, \texttt{cond})$ or $\mathbf{c} \triangleq (loc, \texttt{!cond})$, where `cond` is the condition of the branching instruction at location *loc*, and $t \leadsto_P \mathbf{c}$ if $t$ reaches some $(loc, S)$ such that `cond` evaluates to *true* (resp. *false*) in $S$.

Finally, given a test suite *TS* and a set $\mathbf{C}$ of test objectives, the *coverage score* of *TS* w.r.t. $\mathbf{C}$ is the ratio of the number of test objectives in $\mathbf{C}$ covered by *TS* to its cardinality $|\mathbf{C}|$. The coverage score of *TS* w.r.t. a coverage criterion $\mathbb{C}$ is defined as its coverage score w.r.t. the set $\mathbf{C} = \mathbb{C}(P)$. We also designate it as the $\mathbf{C}$ score (resp. $\mathbb{C}$ score) of *TS* to emphasize the underlying set of test objectives (resp. coverage criterion).

### 3.2. Coverage criteria

This section defines the standard coverage criteria (a.k.a adequacy criteria) used throughout the paper and their associated notions of covering. We follow the classification of Ammann and Offutt [25].

**Control-flow graph and call graph coverage criteria.** The Statement Coverage (**SC**) criterion (a.k.a. Instruction Coverage) requires a test suite to *cover,* that is, to reach, each statement of the program under test, while Decision Coverage (**DC**, a.k.a. Branch Coverage) requires a test suite to *cover* each branch of the program, that is, to activate both true and false branches of each program decision. Function Coverage (**FC**) is a restricted form of **SC**, requiring only to reach all function entrypoints.

**Logic expression coverage criteria.** The three simplest criteria of this family are **CC**, **DCC** and **MCC**. Condition coverage (**CC**) requires to activate both true and false values for each of the atomic conditions in the predicates appearing in any program decision point (e.g. conditional or loop predicates). Decision-Condition Coverage (**DCC**) requires to satisfy both **DC** and **CC**. Multiple-Condition Coverage (**MCC**) requires to activate all combinations of truth values of atomic conditions in each decision. **MCDC** is a family of more intricate logic expression coverage criteria [42], well-known for being required for certification of aeronautic software. In a nutshell, the **MCDC** criteria require to demonstrate that each single atomic condition, alone, can influence the value of the whole decision. In this work, we focus on two **MCDC** criteria, General Active Clause Coverage (**GACC**) and General Inactive Clause Coverage (**GICC**). Notice that while **MCDC** is much more complex to cover than **DCC**, it requires a number of tests only linear in the number of atomic conditions, whereas **MCC** may require an exponential number of tests. Finally, the Implicant Coverage (**IC**) and Unique True Point Coverage (**UTPC**) criteria also aim at covering different behaviors of the predicates appearing at the program decision points, but considering that these predicates have been first syntactically normalized into Disjunctive Normal Form (DNF). The reader can find a detailed definition of these criteria in [25]. (It will be explicitly expressed using labels below in the sketch of proof for Theorem 1.)

**Mutation criteria.** In mutation testing [43], test objectives consist of *mutants*, i.e. slight syntactic modifications of the program under test $P$. In the strong mutation setting **M**, a mutant $M$ is *covered* (or *killed*) by a test datum $t$ if the output of $P(t)$ differs from the output of $M(t)$. In the *weak mutation setting* **WM** [49], a mutant $M$ is covered by $t$ if the internal states of $P(t)$ and $M(t)$ differ from each other right after the mutated location (see Fig. 5). **M** is a very powerful coverage criterion [24,60]. While less powerful in theory, **WM** is almost equivalent to **M** in a practical setting [58].

Mutation testing is parameterized by a set of *mutation operators* $O$. An (atomic) mutation operator $op \in O$ is a function mapping a program $P$ into a finite set of well-defined programs (mutants), such that $P$ differs from each mutant $M$ in only one location. We denote $\mathbf{M}_O$ and $\mathbf{WM}_O$ the strong and weak mutation criteria restricted to mutants created through operators in $O$.

**Black-box criteria.** The Input Domain Coverage criterion (**IDC**) assumes a partition of the input domain $D$ of $P$ given as disjoint predicates $\varphi_1, \ldots, \varphi_k$, and consists of considering one input for each $\varphi_i$.

### 3.3. Symbolic execution

We recall here a few basic facts about Symbolic Execution (SE) [23,53].

Let us consider again the program $P$ presented in Section 3.1 and its input variables $V$, defined over domain $D$. Let us also consider an *execution path* $\sigma$ in the control-flow graph of $P$, i.e. a path in the graph linking the starting node of $P$ to

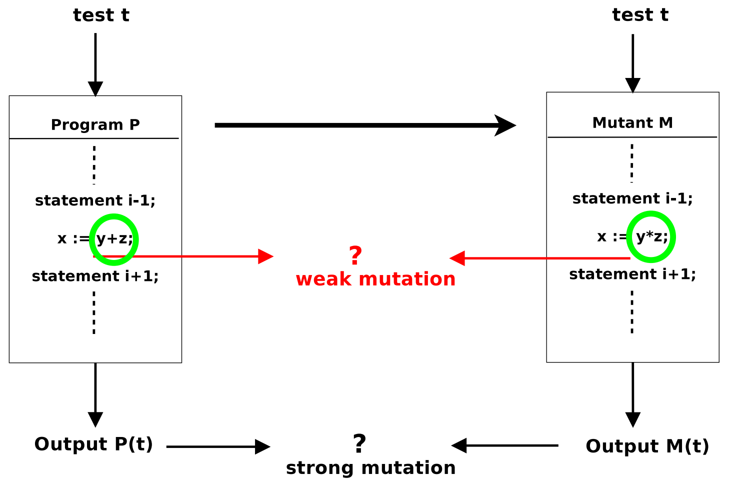


**Fig. 5.** Strong and weak mutations.

```
Input: a program $P$ and a finite set of its execution paths $Paths(P)$
Output: a test suite $TS$, i.e. a set of pairs $(t,\sigma)$, such that $\forall$ feasible $\sigma \in Paths(P)$, $\exists(t,\sigma) \in TS$, $P(t) \leadsto_P \sigma$
$TS := \emptyset$;
$S_{paths} := Paths(P)$;
while $S_{paths} \neq \emptyset$ do
    choose $\sigma \in S_{paths}$;  $S_{paths} := S_{paths} \setminus \{\sigma\}$;
    compute path constraint $\phi_\sigma$ for $\sigma$;
    switch Solve($\phi_\sigma$) do
        case sat($t$) do $TS := TS \cup \{(t,\sigma)\}$;
        case unsat do skip;
    end
end
return $TS$;
```

**Algorithm 1:** Symbolic execution algorithm.

one of its intermediate or exiting nodes. The goal of SE is then to generate an input valuation $t_\sigma \in D$ so that $P(t_\sigma)$ covers (i.e. activates) $\sigma$. The key insight of SE is that, if $P$ is deterministic, it is possible to compute a *path constraint* $\phi_\sigma$ for $\sigma$ such that for any input valuation $t \in D$, we have: $t$ satisfies $\phi_\sigma$ iff $P(t)$ covers $\sigma$. Indeed, such a path constraint $\phi_\sigma$ can be built by virtually executing $P$ over symbolic inputs and aggregating the constraints that arise over these symbolic inputs, as one forces the execution to follow path $\sigma$. For example, executing the program of Fig. 1 over the two symbolic inputs `a` and `b` and forcing the execution to follow the path along the then branches of the two conditional statements produces the path constraint `a > 0` $\wedge$ `b > 0`.

Once the path constraint is built, it is solved using an off-the-shelf constraint solver, yielding the expected input valuation $t_\sigma$ or proving that the path is infeasible if the constraint is unsatisfiable. In practice, the path constraint must sometimes be under-approximated, as $P$ might contain statements (like calls to external libraries) from which the corresponding constraints cannot be easily extracted. Moreover, SE requires the availability of an (efficient) solving procedure for the conditions of the path constraint. These two issues have nevertheless been strongly alleviated during the last two decades with (a) the rise of Dynamic Symbolic Execution (DSE), which interleaves concrete and symbolic execution and uses the dynamically collected data to suggest better approximations for the path constraints, and (b) the development of fast constraint solvers based on a Satisfiability Modulo Theories (SMT) approach [22].

Nowadays, DSE is studied and used by a large and dynamic research community and it is the core principle of a wide variety of test input generation tools, successfully applied in the industry. Yet, the large amount of time that may be required to attempt solving the constraints (often written in an undecidable logic), together with the explosion in the number of paths to process, remain the two main bottlenecks faced by the technique when used to generate a test suite *TS* covering a significant set of paths in a real-world application.

A simplified description of the SE process is depicted in Algorithm 1. While highly abstracted, it is sufficient to understand the remainder of the paper. Note that Solve represents a call to the constraint solver, which can either return unsat (no solution found) or sat($t$) (where $t$ is a solution).

## 4. Generic specification of test objectives with labels

### 4.1. Definitions

Given a program $P$, a *label* $l$ is a pair $(loc, \varphi)$ where $loc$ is a location of $P$ and $\varphi$ is a predicate such that:

- $\varphi$ contains only valid expressions using variables visible at location *loc*;
- $\varphi$ contains no side-effect expressions.[2]

An *annotated program* is a pair $\langle P, L\rangle$ where $L$ is a set of labels defined over $P$. A test datum $t$ *covers* a label $l \triangleq (loc, \varphi)$, denoted $t \leadsto_{\langle P,L\rangle} l$, if $t$ reaches (at least once) the location *loc* with some internal state $S$ such that the predicate $\varphi$ is satisfied in $S$.

### 4.2. Simulating coverage criteria using label coverage

Given an annotated program $\langle P, L\rangle$, we define the *label coverage criterion*, denoted **LC**, as the function returning $L$ as the set of test objectives. Thus, a test suite satisfies **LC** if it covers all labels in $L$, denoted $TS \leadsto_{\psi(P)}$ **LC**.

We seek to characterize the power of the **LC** coverage criterion to emulate other criteria. A key notion here is that of *labeling function*. A labeling function $\psi$ maps a program $P$ into an annotated program $\psi(P) \triangleq \langle P, L\rangle$.

**Definition 1.** A coverage criterion **C** *can be simulated by* **LC** if there exists a labeling function $\psi$ that annotates any given program $P$ with labels corresponding to the test objectives derived following **C**, so that, for any test suite *TS*, we have $TS \leadsto_P$ **C** iff $TS \leadsto_{\psi(P)}$ **LC**.

In order to make the test objectives of some of the criteria discussed below directly encodable by labels, we consider in the rest of the paper only *normalized programs*, i.e. programs such that no side-effect occurs in any condition of a branching instruction. This is not a fundamental restriction since any (well-defined) program $P_1$ can be rewritten into a normalized program $P_2$, using intermediate variables to evaluate the side-effect prone conditions outside the branching instruction. For example, `if (x++ <= y && e == f) {...}` becomes `tmp = x++; if (tmp <= y && e == f) {...}`. Notice that similar transformations are automatically performed by the Cil library [62] frequently used by DSE tools for C programs [68,73].

**Basic graph and logic expression coverage criteria.** To simulate **SC** and **FC**, we add one label with a true predicate, respectively, before each statement of the program and at the beginning of each function body. To emulate **DC**, **CC**, **DCC** and **MCC**, we introduce one label per truth value to cover before any decision in $P$. Some illustrating examples are given in Fig. 6. For instance, for **CC** the corresponding labeling function $\psi_{\mathbf{CC}}$ inserts labels that enforce coverage of both truth values of the two atomic conditions `x==y` and `a<b`.

**Theorem 1.** *The coverage criteria **SC**, **DC**, **CC**, **DCC** and **MCC** can be simulated by **LC**.*

**Sketch of proof.** We need to define a suitable labeling function for any of the considered criteria. For **SC**, we choose the labeling function $\psi_{\mathbf{SC}}$ adding all labels of the form $(loc, true)$, where *loc* is any location of $P$. Given a test suite *TS*, $TS \leadsto_P$ **SC** iff *TS* can reach any *loc* of $P$ iff *TS* covers any $(loc, true)$ iff $TS \leadsto_{\psi_{\mathbf{SC}}(P)}$ **LC**. We conclude that **SC** can be simulated by **LC**. The reasoning is similar for **FC**. Other criteria are handled similarly. The labeling function $\psi_{\mathbf{DC}}$ adds the set of all $(loc, \varphi)$ and $(loc, \neg\varphi)$, where *loc* contains a conditional statement with condition $\varphi$. $\psi_{\mathbf{CC}}$ adds the set of all $(loc, a_i)$ and $(loc, \neg a_i)$, where *loc* contains a conditional statement whose atomic conditions are exactly the $a_i$. $\psi_{\mathbf{DCC}}$ adds the union of labels added by $\psi_{\mathbf{DC}}$ and $\psi_{\mathbf{CC}}$. $\psi_{\mathbf{MCC}}$ adds the set of all $(loc, \bigwedge_i \bar{a}_i)$, where *loc* contains a conditional statement whose atomic conditions are the $a_i$, and $\bar{a}_i$ denotes either $a_i$ or $\neg a_i$. □

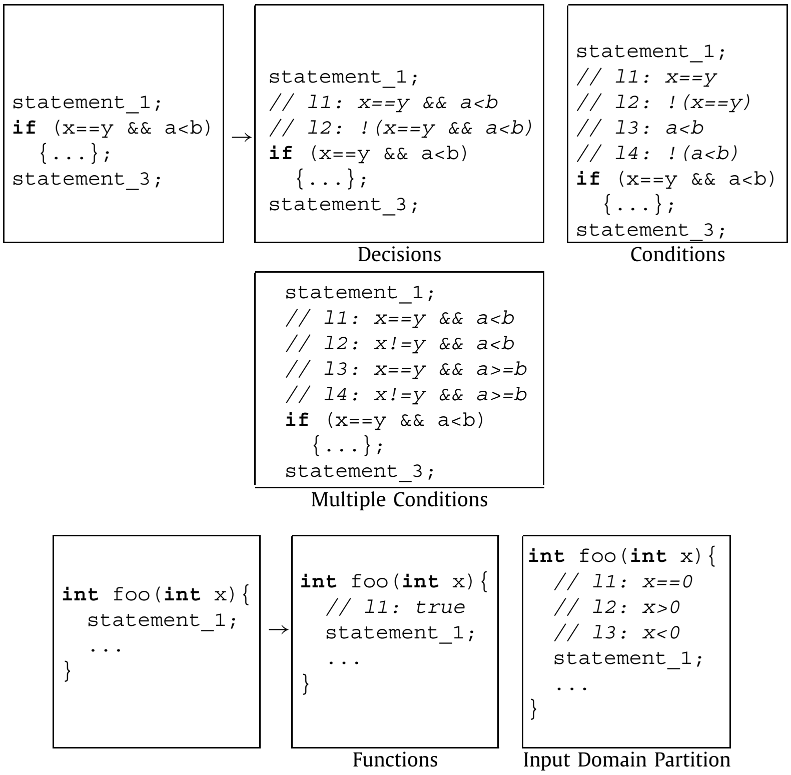
**Advanced logic expression coverage criteria.** Pandita et al. [67] show that **GACC** (and thus **GICC**) can be simulated through additional branches to cover, which can be directly specified in terms of labels, as we did for **DC**. For **IC** and **UTPC**, we introduce one label per truth value to cover in the DNF of each decision predicate in $P$.

**Theorem 2.** *The coverage criteria **GACC**, **GICC**, **IC** and **UTPC** can be simulated by **LC**.*

**Sketch of proof.** Let us consider a predicate $p$ in $P$ that involves $n$ atomic conditions $c_1, \dots, c_n$. **GACC** requires that for each clause $c_i$, the test suite triggers two distinct evaluations of $p$: one execution $A$ where $c_i$ is true, one execution $B$ where $c_i$ is false, and both such that the truth value of $c_i$ impacts the truth value of the whole predicate, i.e.:

$$p(c_1^A, \dots, c_{i-1}^A, true, c_{i+1}^A, \dots, c_n^A) \neq p(c_1^A, \dots, c_{i-1}^A, false, c_{i+1}^A, \dots, c_n^A)$$

[2] We choose to forbid side-effect expressions in label predicates for practical reasons, as it would make the implementation of our testing framework more complex. For example, measuring label coverage in a safe way would require to sandbox the side-effects occurring during label predicate evaluation or to undo them after it.



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

→

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

Decisions

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

Conditions

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

Multiple Conditions

```
int foo(int x){
  statement_1;
  ...
}
```

→

```
int foo(int x){
  // l1: true
  statement_1;
  ...
}
```

Functions

```
int foo(int x){
  // l1: x==0
  // l2: x>0
  // l3: x<0
  statement_1;
  ...
}
```

Input Domain Partition

**Fig. 6.** Simulating standard coverage criteria with labels.

for execution $A$ and:

$$p(c_1^B, \dots, c_{i-1}^B, \mathit{false}, c_{i+1}^B, \dots, c_n^B) \neq p(c_1^B, \dots, c_{i-1}^B, \mathit{true}, c_{i+1}^B, \dots, c_n^B)$$

for execution $B$. For each clause $c_i$ this requirement can be encoded in two atomic labels: $(loc_p, l_{i,A})$ and $(loc_p, l_{i,B})$ with

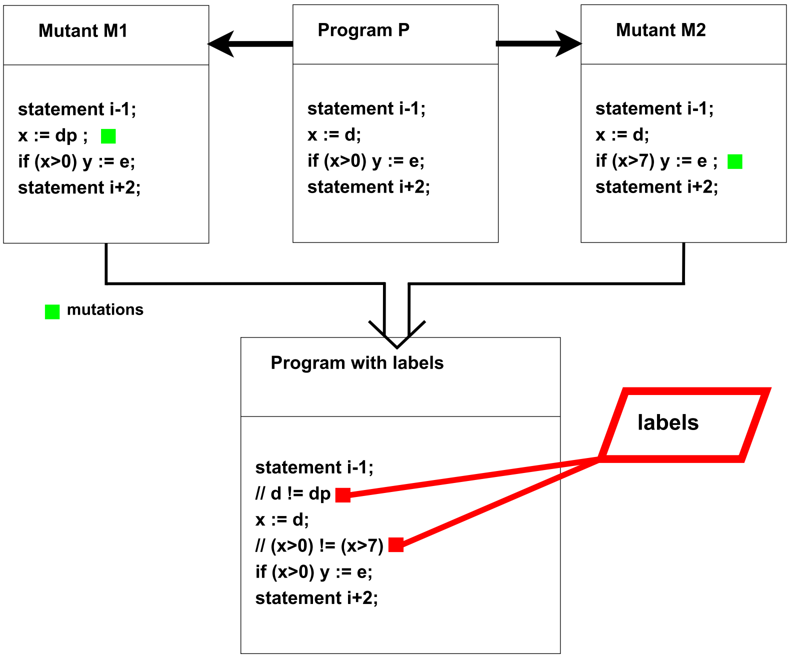
$$\begin{aligned}
loc_p \equiv &\quad \text{the location of predicate } p,\\
l_{i,A} \equiv &\quad c_i \;\wedge\; (\; p(c_1, \dots, c_{i-1}, \mathit{true}, c_{i+1}, \dots, c_n)\\
&\quad \neq p(c_1, \dots, c_{i-1}, \mathit{false}, c_{i+1}, \dots, c_n)\;),\\
l_{i,B} \equiv &\quad \neg c_i \;\wedge\; (\; p(c_1, \dots, c_{i-1}, \mathit{false}, c_{i+1}, \dots, c_n)\\
&\quad \neq p(c_1, \dots, c_{i-1}, \mathit{true}, c_{i+1}, \dots, c_n)\;).
\end{aligned}$$

Similarly, to encode **GICC**, for each clause $c_i$ of a predicate $p$ one needs to define four atomic labels: $(loc_p, l_{i,A})$, $(loc_p, l_{i,B})$, $(loc_p, l_{i,C})$ and $(loc_p, l_{i,D})$ with

$$\begin{aligned}
l_{i,A} \equiv &\quad c_i \wedge p(c_1, \dots, c_n)\\
l_{i,B} \equiv &\quad \neg c_i \wedge p(c_1, \dots, c_n)\\
l_{i,C} \equiv &\quad c_i \wedge \neg p(c_1, \dots, c_n)\\
l_{i,D} \equiv &\quad \neg c_i \wedge \neg p(c_1, \dots, c_n)
\end{aligned}$$

For the next criterion, for a predicate $p$, we consider the disjunctive normal form (DNF) of $p$ and of its negation $\neg p$: $\mathrm{dnf}(p) = \bigvee_i imp_i^{\mathrm{dnf}(p)}$ and $\mathrm{dnf}(\neg p) = \bigvee_k imp_k^{\mathrm{dnf}(\neg p)}$. (The formulas $imp_i^{\mathrm{dnf}(p)}$ and $imp_k^{\mathrm{dnf}(\neg p)}$ are called *implicants* of these DNFs.) To encode Implicant Coverage (**IC**), one needs to define, for each predicate $p$:

- a label $(loc_p, imp_i^{\mathrm{dnf}(p)})$ for any implicant $imp_i^{\mathrm{dnf}(p)}$ of the DNF of $p$,
- a label $(loc_p, imp_k^{\mathrm{dnf}(\neg p)})$ for any implicant $imp_k^{\mathrm{dnf}(\neg p)}$ of the DNF of $\neg p$.



```
statement i-1;
x := dp ; 🟩
if (x>0) y := e;
statement i+2;
```

```
statement i-1;
x := d;
if (x>0) y := e;
statement i+2;
```

```
statement i-1;
x := d;
if (x>7) y := e ; 🟩
statement i+2;
```

```
statement i-1;
// d != dp 🟥
x := d;
// (x>0) != (x>7) 🟥
if (x>0) y := e;
statement i+2;
```

**Fig. 7.** Simulating weak mutants with labels.

Finally, to encode Unique True Point Coverage (**UTPC**), one needs to define, for each predicate $p$:

- a label $(loc_p, imp_i^{\text{dnf}_{min}(p)} \wedge \bigwedge_{j \neq i} \neg imp_j^{\text{dnf}_{min}(p)})$ for each implicant $imp_i^{\text{dnf}_{min}(p)}$ of the minimal DNF[3] $\text{dnf}_{min}(p)$ of $p$,
- a label $(loc_p, imp_k^{\text{dnf}_{min}(\neg p)} \wedge \bigwedge_{l \neq k} \neg imp_j^{\text{dnf}_{min}(\neg p)})$ for each implicant $imp_k^{\text{dnf}_{min}(\neg p)}$ of the minimal DNF $\text{dnf}_{min}(\neg p)$ of $\neg p$.

The reader can easily check that these labels precisely represent the required test objectives according to the definition of these criteria in [25]. □

**Mutation testing.** We now consider an even more involved coverage criterion, namely weak mutations, and show that a well-defined part of **WM** can be simulated by **LC**. We consider only atomic mutations operators that can affect either a left-hand side expression (lhs), an expression or a condition. This is a very generic model of mutations, encompassing all standard operators [25], as well as deletion of assignments, since the replacement of an assignment `x:=exp` by `x:=x` models its deletion. Finally, we restrict ourselves to mutation operators neither affecting nor introducing side-effect expressions (in particular, calls to side-effect prone functions). We refer to such operators as *side-effect free mutation operators*.

**Theorem 3.** *For any finite set $O$ of side-effect free mutation operators, **WM**$_O$ can be simulated by **LC**.*

**Sketch of proof.** We have to define a suitable labeling function. For simplicity, let us consider first a single mutation operator $op \in O$. The main idea is to introduce *one label for each mutant* $M$ created by $op$, so that covering the label is equivalent to distinguishing $M$ from $P$ *once the modified location has been reached*. This transformation is depicted in Fig. 7. Let us consider a mutant $M$ differing from $P$ only at location $loc$. We consider three cases, depending on the modification introduced by $op$:

- $lhs \ := \ expr$ becomes $lhs \ := \ expr'$: we add label $l \triangleq (loc, expr \neq expr')$. We must prove that $t \leadsto_P M$ iff $t \leadsto_{\psi(P)} l$. Note that $t \leadsto_{\psi(P)} l$ iff $t$ reaches $loc$ with an internal state such that $expr$ and $expr'$ evaluate to different values. This is equivalent to say that $P(t)$ and $M(t)$ are in different internal states right after $loc$, which corresponds by definition t $t \leadsto_P M$.

[3] A DNF is minimal if (a) no implicant can be omitted, and (b) no subterm of an implicant can be omitted. We refer the reader to [25] for detailed definitions.

- `if (`$cond$`)` becomes `if (`$cond'$`)`: we add label $l \triangleq (loc, cond \oplus cond')$, where $\oplus$ is the xor-operator. We follow the same line of reasoning as in the previous case. The $\oplus$ operator ensures that $P(t)$ and $M(t)$ will not follow the same branching condition.
- $lhs$ `:=` $expr$ becomes $lhs'$ `:=` $expr$: we add label $l \triangleq (loc, \alpha(lhs) \neq \alpha(lhs') \wedge (lhs \neq expr \vee lhs' \neq expr))$, where $\alpha(x)$ denotes the memory location ($\approx$ address) of $x$, not its value. For example, in C the memory location is given by the `&` operator. This case requires a little bit more explanation. In order to observe a difference between $P(t)$ and $M(t)$ right after the mutated location, we need first that $lhs'$ and $lhs$ refer to different memory locations (which is not always obvious in the case of aliasing expressions). Moreover, no difference can be observed if both locations had the assigned value, i.e. if the old value of $lhs$ and the old value of $lhs'$ were equal to $expr$ in $P(t)$ before the assignment. To observe the difference, at least one of them should be modified by the assignment. This is exactly what $l$ encodes.

By iterating this technique for all considered mutation operators $op \in O$, we obtain the desired labeling function. □

The subset of mutations we have been considering so far is limited to (1) atomic mutations and (2) side-effect free operators. The first restriction is not a major issue as atomic mutations have been observed to be almost as powerful as high-order mutations [57]. The second restriction has two sides: (2.a) it forbids mutation operators *introducing* side-effects, for example mapping `x` to `x++`, and (2.b) it forbids to mutate a side-effect prone expression. Restriction (2.a) is not severe: it encompasses operators ABS, ROR, AOR, COR and UOI [25], which have been shown mostly equivalent to much larger sets of operators [59,72]. It is left as an open question to quantify more precisely what is lost with restriction (2.b).

**Black-box criteria.** Assuming a partition of the input domain $D$ of program $P$ given as disjoint predicates $\varphi_1, \ldots, \varphi_k$, the **IDC** criterion requires one input $t_i$ for each $\varphi_i$. The corresponding labeling function adds all labels of the form $(loc_0, \varphi_j)$, where $loc_0$ is the entry point of $P$. The approach is independent of the way the partition is obtained, covering both interface-based and functionality-based partitions [25, Chap. 4]. Fig. 6 illustrates the approach with $k = 3$, $\varphi_1 \equiv (x == 0)$, $\varphi_2 \equiv (x > 0)$ and $\varphi_3 \equiv (x < 0)$.

### 4.3. *Specifying other useful test objectives with labels*

While test generation tools are often used to generate coverage adequate test suites, they have also been applied in other use cases, like e.g. runtime failure detection [21] or patch testing [20]. Test objectives corresponding to *run-time failures* such as those implicitly searched for in active testing or assertion-based testing [40,47,52] can be easily captured by labels, including division by zero, out-of-bound array accesses or null-pointer dereference. Typically, any error-prone instruction at location $loc$ with a precondition $\varphi_{\text{safe}}$ will be tagged by a label $(loc, \neg\varphi_{\text{safe}})$. Test objectives corresponding to reaching code zones *affected by a patch* can be encoded by labels with a true predicate at the entrance of each basic bloc modified by the patch.

### 4.4. *Limitations of label expressiveness*

The following classes of test objectives cannot be *directly* encoded through labels [19]:

- objectives constraining paths rather than program locations (e.g. data-flow or prime paths coverage criteria [25]),
- objectives relating different paths (e.g. **MCDC** criteria other than **GACC** and **GICC**, hyperproperties), possibly in slightly different programs (i.e. strong mutations).

While the first class of criteria can be encoded by labels with the help of additional instrumentation (see e.g. [18] for data-flow criteria), for the others no simple encoding has been found yet. As already pointed out, weak mutations with side-effect operators are also outside the direct scope of labels. When no exact simulation is known, labels can still be used for approximations. For example, even the most intricate **MCDC** criterion (a.k.a. **RACC**) can be upper-approximated (with **MCC**) or lower-approximated (with **GACC**).

An extension of labels, named *hyperlabels*, is presented in Section 10.1. Hyperlabels [8,9] provide combination operators over labels, yielding a very expressive framework for the specification of test objectives. While standard labels are restricted to state-reachability constraints (the test datum must reach a specific state), hyperlabels can express test objectives defined over trace reachability (the test datum must follow a particular sequence of states) or even hyper-reachability (constraints are here expressed over finite sets of traces, typically pairs).

## 5. Infeasible label detection

A significant proportion of the labels in an annotated program can turn out to be infeasible, i.e. no input can satisfy them. Infeasible labels are a threat to the efficiency of label-driven test generation, because a significant part of the test budget might be lost trying to cover them. In this section, we introduce a sound approach to prune out infeasible labels.

### 5.1. Definitions

We first formally define infeasible labels.

**Definition 2.** Given a label $l \triangleq (loc, \varphi)$ in an annotated program $\langle P, L\rangle$, we say that $l$ is infeasible if there is no input datum $t$ such that $t \rightsquigarrow_{\langle P,L\rangle} l$.

Given a program $P$, an *assertion* is a pair $(loc, \varphi)$ where *loc* is a location of $P$ and $\varphi$ is a predicate that contains only valid and side-effect free expressions.

**Definition 3.** An assertion of $P$ is **valid** iff for any test datum $t$ and for any internal state $S$ of $P$ such that $t$ reaches *loc* with internal state $S$, we have that $\varphi$ evaluates to true in $S$.

**Definition 4.** Given a label $l \triangleq (loc, \varphi)$ in an annotated program $\langle P, L\rangle$, the **opposite assertion** of $l$ is the assertion $(loc, \neg\varphi)$.

### 5.2. Reducing label infeasibility to assertion validity

**Lemma 4.** *A label is infeasible iff its opposite assertion is valid.*

**Sketch of proof.** If a label $l \triangleq (loc, \varphi)$ is infeasible, then for any test datum $t$ of $P$ that reaches *loc* with some internal state $S$, we have that $\varphi$ evaluates to false in $S$. As a consequence, $\neg\varphi$ evaluates to true in $S$. Thus, by definition, the opposite assertion of $l$ is valid. The reverse can be proven by a similar reasoning. □

### 5.3. A practical approach to detecting infeasible labels

Assertion validity is a very common kind of *safety properties* and many assertion checker tools are available [18,27], relying on various formal techniques such as weakest-precondition or value analysis, as well as model-checking. By Lemma 4, a natural approach to detect if a label is infeasible is to send its opposite assertion to an off-the-shelf assertion checker: if the checker is able to prove that the assertion is valid, then we know that the label is infeasible. As assertion checking is an undecidable and complex problem, the assertion checker may time out or return no answer, so that our infeasible label detection approach is only partial.

## 6. Label coverage measurement and label-driven symbolic execution

### 6.1. Naive approach

Given an annotated program $\langle P, L\rangle$, we seek for automatic methods for: (1) computing the **LC** score of a given test suite *TS*, and (2) deriving a test suite achieving high **LC** score. We propose first a black-box approach, reusing standard automatic testing tools through a *direct instrumentation* of $P$. This technique underlies previous works aiming at extending DSE coverage abilities [40,47,50,52,66,67,76]. While it allows for cheap **LC** score computation, it is far from efficient for automated test generation (abbreviated as ATG below), mainly because of an exponential blow-up of the path space of the program.

#### 6.1.1. Direct instrumentation

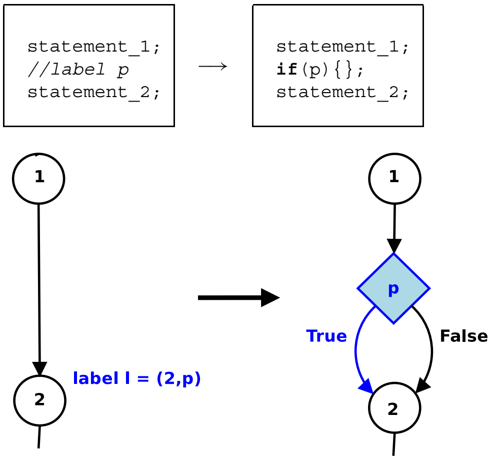
The *direct instrumentation* $P'$ for $\langle P, L\rangle$ consists in inserting for each label $l \triangleq (loc, \varphi) \in L$ a new branching instruction $I$: `if (`$\varphi$`) {};` such that all instructions leading to *loc* in $P$ lead to $I$ in $P'$, and $I$ leads to *loc*. The transformation is depicted in Fig. 8. When different labels are attached to the same location, the new instructions are chained together in a sequence ultimately leading to *loc*.

Let us denote by **NTD** the set of test objectives over $P'$ requiring to cover all **N**ew **`T`**`hen`-**D**ecisions introduced by this transformation. Direct instrumentation is obviously sound w.r.t. **LC** in the following sense.

**Theorem 5** *(Soundness). Given an annotated program $\langle P, L\rangle$, its direct instrumentation $P'$ and a test suite TS, we have: $TS \rightsquigarrow_{\langle P,L\rangle}$* **LC** *iff $TS \rightsquigarrow_{P'}$* **NTD**.

This is interesting for both **LC** score computation and ATG. Indeed, any ATG tool run on $P'$ will produce a test suite *TS* covering **LC** for $\langle P, L\rangle$ as soon as *TS* covers all branches of interest in $P'$. Besides, a slightly modified version of direct instrumentation, updating coverage information in the new `then`-branches, allows efficient coverage score computation.

**Theorem 6.** *Given an annotated program $\langle P, L\rangle$, its direct instrumentation $P'$ and a test suite TS, then the* **LC** *score of TS can be computed in time bounded by $|TS| \cdot maxtime(\{P'(t) | t \in TS\})$.*



```
statement_1;
//label p
statement_2;
```

```
statement_1;
if(p){};
statement_2;
```

**Fig. 8.** Direct instrumentation $P'$.

```
int numPos(int a, int b) {
  int n = 0;
  if( a > 0 ) {
    // l1: a == 5
    n++;
  }
  if( b > 0 ) n++;
  // l2: n > 0
  return n;
}
```

**Fig. 9.** Function `numPos` with two labels.

Note that by computing the maxtime over $P'$ in the above formula, we implicitly include the overhead of evaluating labels within the code. The expectation is that only a small fraction of the labels in the program are evaluated in any path, so that $maxtime(P')$ is substantially smaller than $|L| \times maxtime(P)$. It follows from Theorems 3 and 6 that coverage measurement of the side-effect free subset of weak mutation (**WM**) can then be efficient in practice: rather than re-running program execution for each mutant, we can measure a relatively small number of labels for a given test case.

### 6.1.2. Discussion

The direct instrumentation, while useful for label coverage measurement, is inefficient for ATG using symbolic execution, because of two main issues that we illustrate on the function `numPos` annotated with two labels as shown in Fig. 9.

The paths of this program and those resulting from its direct instrumentation $P'$ are shown in Fig. 10 (a) and (b). The first issue is that the initial program has only 4 paths, while the direct instrumentation leads to 12 more intricate paths (some of which being infeasible). In particular, all the feasible paths covering the second label contain a constraint coming from the condition of the first label, that is, the condition `a=5` or its negation. Not present in the initial program logic, this constraint is irrelevant for covering subsequent branches and uselessly increases the size of the path constraint to be solved during DSE.

The second issue is that running DSE on $P'$ will lead to covering the second label five times since five feasible partial paths lead to the condition `n>0` in $P'$, and five test cases will be generated to cover it (e.g. $(a, b) = (5, 1)$, $(5, 0)$, $(1, 1)$, $(1, 0)$ and $(0, 1)$). In contrast, a minimal test suite covering the two labels is the single case $(a, b) = (5, 1)$.

To synthesize, the two issues of direct instrumentation that make it inefficient for ATG are:

(†) $P'$ is too complex: it exhibits many more paths than $P$, most of them being unduly complex for covering the labels we are targeting.

(‡) DSE will naturally produce test suites that cover the same labels several times, requiring substantially more analysis effort than necessary.

Let us formalize the first point (†) hereafter. We consider two dimensions in which $P'$ is "too complex": the size of the search space, denoted $|Paths(P')|$, and the shape of the paths in $Paths(P')$. Let us call *label constraints* the conditions of all additional branches $\varphi$ and $\neg\varphi$ introduced in $P'$ compared to $P$.

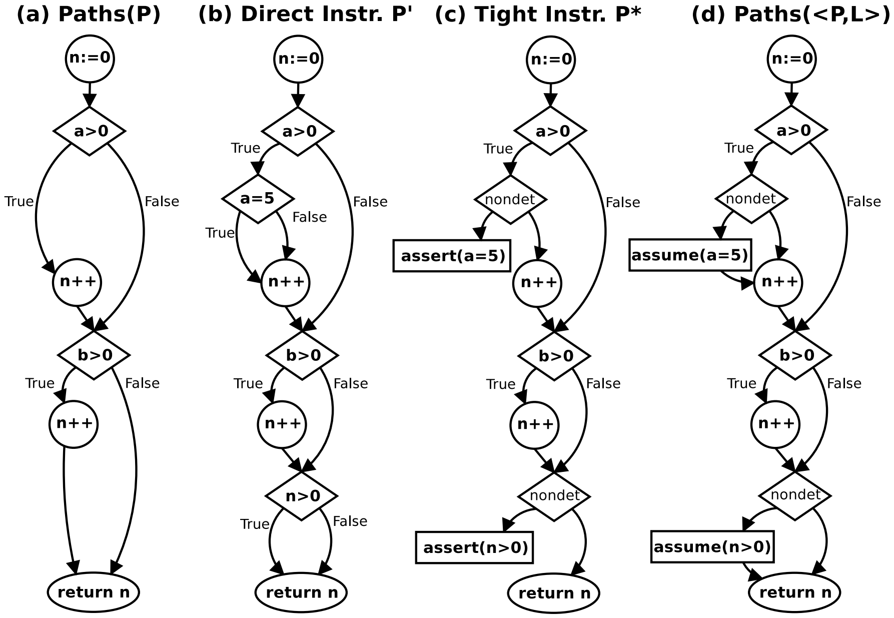


**Fig. 10.** *Paths*($\langle P, L\rangle$) compared to the paths of $P$, its direct and tight instrumentation for function `numPos` of Fig. 1 annotated by two labels.

**Theorem 7** *(Non-tightness). Given an annotated program $\langle P, L\rangle$ and its direct instrumentation $P'$, let us assume that Paths($P$) is bounded, that $k$ represents the maximal length of paths in Paths($P$) and that $m$ is the maximal number of labels per location in $P$. Then:*

- *$|Paths(P')|$ can be exponentially larger than $|Paths(P)|$ by a factor $2^{m\cdot k}$;*
- *any $\sigma' \in Paths(P')$ may carry up to $m \cdot k$ (positive or negative) label constraints.*

**Sketch of proof.** A single path $\sigma \in P$ may correspond to up to $2^{m\cdot|\sigma|}$ paths in $P'$, since each label of $P$ creates a branching in $P'$ and at most $m$ such branchings can be found at each step of $\sigma$. Note also that the paths $\sigma' \in P'$ corresponding to $\sigma \in P$ have length bounded by $m \cdot |\sigma|$. Therefore they can pass through up to $m \cdot |\sigma|$ label constraints, while (by definition) $\sigma$ does not pass through any label constraint. □

Both aspects are problematic for symbolic execution: more paths means more requests to a constraint solver, while more constraint-laden paths means more expensive requests.

### 6.2. *Efficient label-driven DSE*

We describe in this section two main ingredients in order to obtain efficient ATG for **LC**: (1) *a tight instrumentation* avoiding all drawbacks of the direct instrumentation, and (2) a strong coupling of label coverage and DSE through *iterative label deletion*.

#### 6.2.1. *Tight instrumentation*

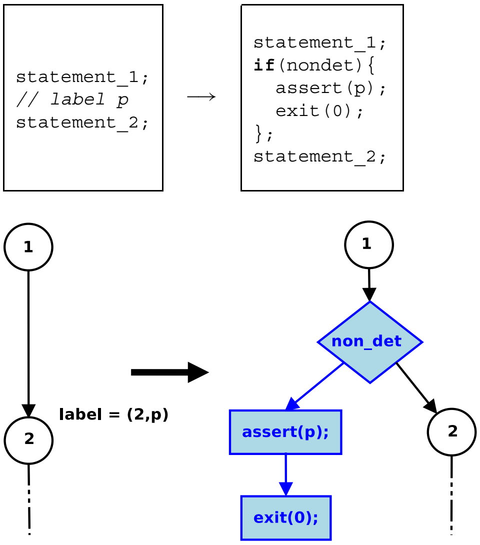
Given a label $l \triangleq (loc, \varphi)$, the key insights behind the tight instrumentation are the following:

- label constraint $\varphi$ is useful only for covering $l$, and should not be propagated beyond that point;
- label constraint $\neg\varphi$ is pointless w.r.t. covering $l$, and should not be enforced in any way.

Keeping these lines in mind, the instrumentation works as depicted in Fig. 11: for each label $(loc, \varphi)$, we introduce a new instruction `if (nondet) {assert(`$\varphi$`); exit};` where `assert(`$\varphi$`)` requires $\varphi$ to be verified, `exit` forces the execution to stop and `nondet` is a non-deterministic choice.[4]

In the resulting instrumented program $P^\star$ (Fig. 11, right column), when an execution reaches *loc*, it gives rise to two execution paths: the first one tries to cover the label by asserting $\varphi$ and *stops right there*, the second one simply follows its execution *as it would do in* $P$, neither $\varphi$ nor $\neg\varphi$ being enforced.

[4] Note that any DSE engine can simulate non-deterministic choices by an additional input array of (symbolic) Boolean values.



```
statement_1;
// label p
statement_2;
```

→

```
statement_1;
if(nondet){
  assert(p);
  exit(0);
};
statement_2;
```

**Fig. 11.** Tight instrumentation $P^\star$.

Let us denote by **NA** the set of test objectives over $P^\star$ requiring to cover all **N**ew `Assert` introduced by the instrumentation with condition evaluating to true (cf. Fig. 11). Alternatively, **NA** comes down to cover all new instructions coming from the tight instrumentation, or to cover all the newly introduced `exit(0)`. We also extend our definition of coverage for **NA** as follows, as $P^\star$ contains non-deterministic choices: $TS \rightsquigarrow_{P^\star}$ **NA** if for each $na \in$ **NA**, there is a test datum $t \in TS$ such that one of the possible executions of $P^\star$ over $t$ covers $na$. Tight instrumentation is sound w.r.t. **LC** in the following sense.

**Theorem 8** *(Soundness). Given an annotated program $\langle P, L\rangle$, its tight instrumentation $P^\star$ and a test suite TS, we have: $TS \rightsquigarrow_{\langle P,L\rangle}$* ***LC*** *iff $TS \rightsquigarrow_{P^\star}$* **NA**.

**Sketch of proof.** We use the notation of Fig. 11. Clearly, if (the execution of) a test datum $t$ can reach a newly introduced `exit(0)` in $P^\star$ (preceded by instruction `assert(p)` which corresponds to location *loc*), then $t$ over $P$ reaches location *loc* with a memory state satisfying $p$, as the newly introduced instructions cannot modify memory states, they can just observe it. Conversely, if a test datum $t$ over $P$ reaches a location *loc* with a memory state satisfying predicate $p$, there is an execution in $P^\star$ where $t$ reaches the corresponding newly introduced `exit(0)`: indeed, it happens when all non-deterministic choices but the one covering the label choose to follow the original program execution, while the remaining one chooses to bifurcate over the `assert(p)` when the execution reaches *loc* with an adequate memory state – which is ensured to happened, as it does over $P$. □

In particular, an ATG tool run on $P^\star$ and producing a test suite $TS$ covering **NA** will also cover the labels of the annotated program $\langle P, L\rangle$. The following result now follows from Theorems 5 and 8.

**Corollary 9.** *Let $\langle P, L\rangle$ be an annotated program with direct instrumentation $P'$ and tight instrumentation $P^\star$. Then both DSE($P^\star$) and DSE($P'$) achieve the same coverage and can be used to cover all coverable labels in $\langle P, L\rangle$.*

Interestingly, tight instrumentation does not show any of the issues reported in Theorem 7. The underlying reasons have been sketched at the beginning of Section 6.2.1 and are depicted in Fig. 12. A single execution path in $P$ going through $n$ labels gives birth up to $2^n$ paths in $P'$ (left column), while it creates at most $n+1$ paths in $P^\star$ (right column). Moreover, each path in $P^\star$ can go through at most one single positive label constraint (because of the `exit` node), while a path $\sigma'$ in $P'$ can carry up to $|\sigma'|$ (positive or negative) label constraints. These results are summarized in Theorem 10.

**Theorem 10** *(Tightness). Given an annotated program $\langle P, L\rangle$ and its instrumented version $P^\star$, let us assume that $Paths(P)$ is bounded, that k is the maximal length of paths in $Paths(P)$ and that m is the maximal number of labels per location in $P$. Then $P^\star$ is tight in the following sense:*

- *$|Paths(P^\star)|$ is linear in $|Paths(P)|$ and $m \cdot k$. More precisely, $|Paths(P^\star)|$ is bounded by $(m \cdot k + 1) \cdot |Paths(P)|$;*
- *any $\sigma \in Paths(P^\star)$ carries at most one label constraint.*

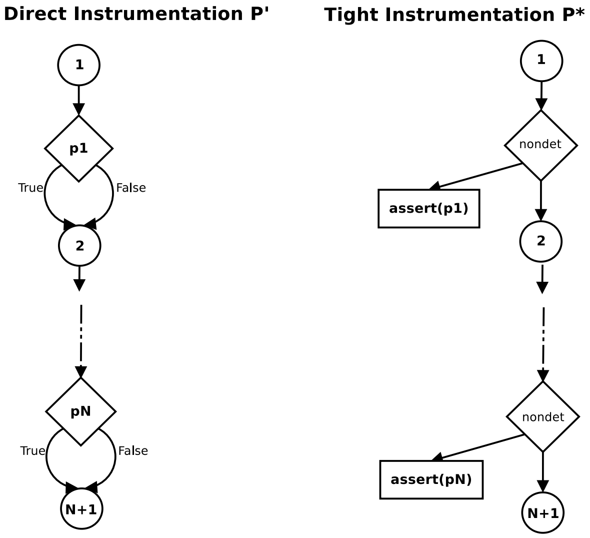


**Fig. 12.** Direct vs. tight instrumentation.

**Sketch of proof.** The main reasons behind this result directly follow from the tight instrumentation, and have already been exposed just before Theorem 10. □

To illustrate the benefits of tight instrumentation on our running example, consider the paths of $P$, $P'$ and $P^\star$ shown in Fig. 10(a), (b) and (c). Instead of 12 paths of $P'$, the tight instrumentation $P^\star$ contains only 9 paths, that is, one additional path for each partial path reaching a label in $P$. Unlike in direct instrumentation, neither of these paths uselessly keeps any constraint for the first label: either the label constraint is the last one in the path and is necessary to cover the label, or it is bypassed without complicating the constraint set for covering subsequent branches.

Theorems 7 and 10 imply that any path-based program analysis like DSE conducted over $P^\star$ will have a much easier task than if conducted over $P'$, since $P^\star$ contains exponentially fewer paths and those paths are simpler. The issue (†) identified in Section 6.1.2 is thus avoided for tight instrumentation.

### 6.2.2. *Iterative label deletion*

We focus now on issue (‡) pointed out in Section 6.1.2. A DSE procedure launched on $P^\star$ tries to cover all paths from $P^\star$, while we are only interested in covering branches corresponding to labels. Especially, it may try to cover (partial) paths ending in an already-covered `assert(`$\varphi$`)`. Whether they fail or not, these computations are redundant since Theorem 8 only requires each new `assert` to be covered once.

For our running example, DSE on the tight instrumentation $P^\star$ will still try to cover the second label four times since four feasible partial paths lead to the condition `n>0` in $P^\star$ (cf. Fig. 10(c)), and will generate three test data to cover it (e.g. $(a, b) = (1, 1)$, $(1, 0)$ and $(0, 1)$).

*Iterative deletion of labels* (IDL) aims at taming issue (‡) by (conceptually) erasing label constraints as soon as they are covered, so that that does not affect the rest of the path search. In practice, this can be implemented by making our program instrumentation keep track of the coverage state of each label during the path exploration by the DSE tool. We depict in Fig. 13 how our tight instrumentation can be modified in such a way. In a nutshell, each label $l$ is associated with a boolean flag $covered_l$ in an external database called $label\_state\_db$, and this flag should be true iff $l$ has already been covered during path exploration. In addition, the value of $covered_l$ in the database can be read or set by using the $is\_covered(label\_state\_db, l)$ and $set\_covered(label\_state\_db, l, value)$ primitives. Before the path exploration starts, flag $covered_l$ is initialized to false. As a consequence, the call to $is\_covered(label\_state\_db, l)$ guarding the access to label $l$ in the modified tight instrumentation will always return false and the DSE tool will be allowed to explore paths leading to $l$ being covered.[5] Yet, as such paths end by a call to $set\_covered(label\_state\_db, l, true)$, as soon as one of them has been explored by the DSE tool, all the subsequent calls to $is\_covered(label\_state\_db, l)$ guarding the access to the label $l$ will always return true and the DSE tool will thus be barred from exploring any additional path leading to $l$ being covered. This behavior corresponds to the label being erased from the program after being covered by the DSE tool.

[5] A reader familiar with DSE may point out that the DSE tool will consider the label state database as another input of the program and try to generate values for the label flags as well. However, this undesired behavior can be easily avoided in most DSE tools by making the state database a *purely concrete* value.

```
statement_k;
if(nondet){
  assert(p);
  exit(0);
}
statement_k+1;
```

⟶

```
statement_k;
if(nondet && !is_covered(label_state_db,l)){
  assert(p);
  set_covered(label_state_db,l,1);
  exit(0);
}
statement_k+1;
```

**Fig. 13.** IDL variant of tight instrumentation $P^\star$.

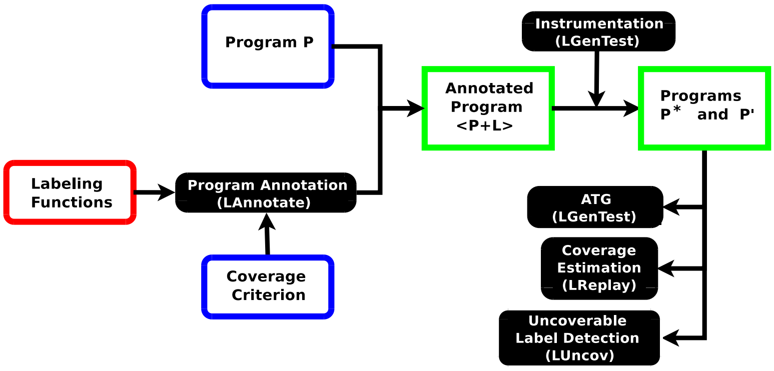

Program P
Instrumentation (LGenTest)
Annotated Program <P+L>
Programs P* and P'
Labeling Functions
Program Annotation (LAnnotate)
Coverage Criterion
ATG (LGenTest)
Coverage Estimation (LReplay)
Uncoverable Label Detection (LUncov)

**Fig. 14.** **LC**-based testing framework.

We denote by DSE*($P^\star$) this combination of tight instrumentation and IDL. Considering only deterministic and sound DSE techniques, the following result holds.

**Theorem 11** *(Relative completeness). Let $\langle P, L\rangle$ be an annotated program, and $P^\star$ its tight instrumentation. Then DSE*($P^\star$) covers as many labels as DSE($P^\star$) does.*

**Sketch of proof.** The main argument here is that discarding a path $\sigma$ passing through an already covered label $l$ in DSE*($P^\star$) can never prevent from covering another not-yet-covered label $l'$, as by construction there always exist a "label free" path $\sigma'$ – similar to $\sigma$ but without label constraints – passing through $l'$. □

## 7. Implementation: the LTest toolset

Putting Sections 4, 5 and 6 together, we see that labels form the basis of a very powerful framework for automated testing, providing various services and handling many different flavors of test objectives in a complete and uniform fashion. Fig. 14 gives an overview of this framework. Starting from a program $P$ and choosing a way **C** to derive test objectives (like a coverage criterion), the labeling function $\psi_{\mathbf{C}}$ creates the **C**-compliant label-annotated program $\langle P, L\rangle$. In particular, predefined labeling functions are available for all common coverage criteria. Assertion checkers can be leveraged on $\langle P, L\rangle$ in order to detect infeasible (or uncoverable) labels. Finally, efficient **LC** score computation can be performed to measure the coverage of a given test suite, and efficient label-driven DSE can be used to cover feasible labels.

### 7.1. Overview of the LTest platform

We have implemented our framework in the context of C programs within the LTest toolset [26], which offers the following services:

**Program annotation:** the LAnnotate module annotates the program with labels according to a chosen coverage criteria.

**Uncoverable label detection:** the LUncov module leverages off-the-shelf assertion checkers to detect infeasible labels. The information is primarily used by other modules, but it can also be exported for external use.

**Coverage estimation:** the LReplay module replays a given test suite and reports its label coverage. Coverage is given as a whole (all test objectives taken into account) and per criterion. Moreover, infeasible and uncovered labels are reported.

**ATG:** the LGenTest module uses DSE*($P^*$) to build a test suite covering the labels. In case a test suite has already been replayed, LGenTest will try to complete the achieved coverage rather than starting from scratch.

The platform currently provides automatic label annotation for the following coverage criteria: decision coverage (**DC**), function coverage (**FC**), condition coverage (**CC**), multiple-condition coverage (**MCC**), weak mutation (**WM**, operators AOR, ROR, COR, ABS), interface-based input domain partition (**IDC**) and general active clause coverage (**GACC**). Moreover, coverage criteria can be combined together, test objectives can be restricted to certain procedures of the program under test and it is possible to add hand-written test objectives.

### *7.2. Technical details*

The four modules LAnnotate, LReplay, LUncov and LGenTest interact through shared information comprising the annotated program and a database mapping each label to its current status, namely: *covered*, *uncoverable*, *unknown* (i.e. neither covered nor proven uncoverable). LAnnotate acts as a front-end and comes with predefined labeling functions for all the criteria supported by the platform. It annotates the program with the corresponding labels according to the criteria chosen by the user and creates the status database. An API is provided for developers to easily write labeling functions for additional criteria. Labels are inserted in the C code as external function calls and can be suppressed on-demand by the C compiler via a simple C macro. Manual edition of labels in the source code and status database is made easy if needed. The three other modules provide user-level services. They can update label statuses and take advantage of them.

The LTest toolkit is built on top of the Frama-C verification platform for C programs [41] whose main analyzers are open source (LGPL). We took advantage of the plugin-based architecture of Frama-C, reusing existing analyses of interest for our needs. In particular, LUncov takes advantage of the assertion checking capabilities of Frama-C to detect infeasible labels. These capabilities are based on the weakest-precondition and value analysis approaches. The Frama-C kernel, LAnnotate, LGenTest and LUncov modules are written in OCaml. The ATG engine of LGenTest relies on an implementation of DSE$^\star$ within the DSE tool PathCrawler [73], written in ECLiPSe/Prolog. The LTest code (except for the LGenTest module) is open-source (LGPL).[6]

## 8. Experimental evaluation

### *8.1. Evaluating label-driven DSE for basic coverage criteria*

**Objectives.** This first batch of experiments have been conducted in order to investigate the following properties of our label-driven DSE approach over basic common coverage criteria:

- the relative gain of our optimizations (cf. Sections 6.2.1 and 6.2.2) w.r.t. direct instrumentation,
- the overhead of lifting DSE to **LC** and the gain in terms of coverage,
- the gain in terms of coverage over random testing.

**Protocol.** We consider 12 standard benchmark programs[7] taken from related works [40,66,64], mainly coming from the Siemens test suite (`tcas`), the Verisec benchmark (`get_tag` and `full_bad` from Apache source code) and MediaBench (`gd` from libgd). We also consider three classes of labels simulating standard coverage criteria of increasing difficulty: **CC**, **MCC** and **WM** (cf. Section 4.2). For **WM**, we use the operators AOIU, AOR, COR and ROR [25] (in a similar way to what is done by MuJava in a Java context [54]), which are considered very powerful in practice [59,72]. In the end, we consider a total of 26 pairs of programs and coverage criteria, discarding the experiments with **CC** and **MCC** for programs without decision predicates involving more than one atomic condition (in this case **CC** and **MCC** indeed come down to **DC**).

We compare the following testing techniques (cf. the columns of Tables 1 and 2): random testing (witness) denoted Random, standard DSE (witness) denoted DSE($P$), standard DSE on direct instrumentation denoted DSE($P'$), standard DSE on tight instrumentation denoted DSE($P^\star$), and DSE with iterative label deletion run on tight instrumentation denoted DSE$^\star$($P^\star$). The DSE engine runs in deterministic mode, generating the same concrete values from one run to the other. All DSE variants are stopped either when a chosen time-out value has been reached or when all the DSE-reachable paths within a chosen upper bound in the total number of loop iterations have been covered. As for the variants of DSE, random testing time includes both test generation and test execution steps. For each benchmark program, random testing is allocated the same time as consumed by DSE$^\star$($P^\star$). To avoid non-representative results, random testing is repeated at least 5 times for each example, and even more if necessary until the average number of generated tests is measured with a relative standard error less than 5%. Time-out for the solver is set to 1 min, time-out for test generation is set to 90 min. Experiments are performed on an Intel Core2 Duo 2.40 GHz, 4 GB of RAM.

For DSE derivatives, we record the following information: number of paths explored by the search, computation time and achieved coverage. The number of paths is a good measure for comparing the complexity of the different search spaces, and therefore to assess both the "cost" of lifting DSE to labels and the benefits of our optimizations. Coverage score together with computation time indicate how practical label-based DSE is. For random testing, we record the allocated computation

[6] LTest source code is available online at https://sites.google.com/view/michaelmarcozzi/software/frama-cltest.

[7] http://micdel.fr/public/ltest/benchmarks-TAP2014.tar.gz.

**Table 1**
Experiments (1/2) for DSE$^\star$ vs DSE and random testing.

| | | | Random<br>(witness) | DSE($P$)<br>(witness) | DSE($P'$) | DSE($P^\star$) | DSE$^\star$($P^\star$) |
|---|---|---|---|---|---|---|---|
| trityp<br>50 loc | CC<br>24$l$ | #paths | 1,500 | 35 | 183 | 83 | 46 |
| | | time | 1.6 s | 1.3 s | 1.6 s | 2.0 s | 1.6 s |
| | | cover | 12/24 | 24/24 | 24/24 | 24/24 | 24/24 |
| | MCC<br>28$l$ | #paths | 1,619 | 35 | 337 | 110 | 66 |
| | | time | 2.1 s | 1.3 s | 1.9 s | 3.0 s | 2.1 s |
| | | cover | 16/28 | 17/28 | 28/28 | 28/28 | 28/28 |
| | WM<br>129$l$ | #paths | 4,301 | 35 | x | 506 | 98 |
| | | time | 5.1 s | 1.3 s | x | 12 s | 5.1 s |
| | | cover | 61/129 | 112/129 | x | 125/129 | 125/129 |
| 4balls<br>35 loc | WM<br>67$l$ | #paths | 1,903 | 7 | **195** | 75 | **23** |
| | | time | 2.1 s | 1.2 s | 1.9 s | 1.1 s | 2.1 s |
| | | cover | 25/67 | 55/67 | 56/67 | 56/67 | 56/67 |
| utf8-3<br>108 loc | WM<br>84$l$ | #paths | 2,551 | 134 | 1,379 | 626 | 313 |
| | | time | 3.8 s | 1.4 s | 4.2 s | 4.3 s | 3.8 s |
| | | cover | 52/84 | 53/84 | 55/84 | 55/84 | 55/84 |
| utf8-5<br>108 loc | WM<br>84$l$ | #paths | 5,396 | 680 | **11,111** | 3,239 | **743** |
| | | time | 8.1 s | 2 s | 40 s | 24 s | 8.1 s |
| | | cover | 53/84 | 80/84 | 82/84 | 82/84 | 82/84 |
| utf8-7<br>108 loc | WM<br>84$l$ | #paths | 23,053 | 3,069 | **81,133** | 14,676 | **3,265** |
| | | time | 35 s | 5.8 s | **576 s** | 110 s | **35 s** |
| | | cover | 53/84 | 80/84 | 82/84 | 82/84 | 82/84 |
| tcas | CC<br>10$l$ | #paths | 3,096 | 2,787 | 3,508 | 3,508 | 2,815 |
| | | time | 3.4 s | 2.9 s | 3.6 s | 4.9 s | 3.4 s |
| | | cover | 10/10 | 10/10 | 10/10 | 10/10 | 10/10 |
| | MCC<br>12$l$ | #paths | 3,182 | 2,787 | 3,988 | 3,988 | 3,059 |
| | | time | 3.9 s | 2.9 s | 4.2 s | 5.2 s | 3.9 s |
| | | cover | 11/12 | 10/12 | 11/12 | 11/12 | 11/12 |
| tcas' | WM<br>111$l$ | #paths | 20,347 | 4,420 | **300,213** | 20,312 | **6,014** |
| | | time | 27 s | 5.6 s | **662 s** | 120 s | **27 s** |
| | | cover | 90/111 | 88/111 | 101/111 | 101/111 | 101/111 |
| replace<br>100 loc | WM<br>79$l$ | #paths | 11,747 | 866 | **87,498** | 6,420 | **2,347** |
| | | time | 14 s | 2 s | **245 s** | 64 s | **14 s** |
| | | cover | 68/79 | 69/79 | 70/79 | 70/79 | 70/79 |
| full_bad<br>219 loc | CC<br>16$l$ | #paths | 5,180 | 2,563 | 5,148 | 5,129 | 3,209 |
| | | time | 7 s | 5 s | 8 s | 14 s | 7 s |
| | | cover | 9/16 | 12/16 | 12/16 | 12/16 | 12/16 |
| | MCC<br>39$l$ | #paths | 9,393 | 2,563 | 12,360 | 12,296 | 7,043 |
| | | time | 19 s | 5 s | 19 s | 32 s | 19 s |
| | | cover | 17/39 | 24/39 | 24/39 | 24/39 | 24/39 |
| | WM<br>46$l$ | #paths | 9,425 | 2,563 | 19,336 | 10,610 | 5,414 |
| | | time | 19 s | 5 s | 35 s | 40 s | 19 s |
| | | cover | 29/46 | 34/46 | 34/46 | 34/46 | 34/46 |

TO: time-out (5,400 s), x: crash due to a bug in the underlying solver.

time, the average number of tests generated and executed (indicated in the “#paths” rows of Tables 1 and 2), and the average coverage score.

**Results.** Results for our 26 examples are presented in Tables 1 and 2, and summaries on overhead and achieved coverage can be found in Tables 3 and 4. First, note that when no time-out occurs, direct instrumentation and both variants of tight instrumentation achieve the same coverage, and that this coverage is high (>90% on 18/26 examples). We also observe that direct instrumentation yields a significant overhead, confirming previous work [50]: DSE($P'$) has four time-outs (TO) while DSE($P$) has none, the time-overhead goes up to 122x (excluding TO), and the blow-up of the path-space reaches 50x.

On the other hand, tight instrumentation DSE$^\star$($P^\star$) yields only a very reasonable overhead w.r.t. standard DSE: no time-out is reported, the time-overhead is kept under 7x with an average of 2.15x, and the path-space growth is limited to 3x. On some examples, tight instrumentation performs remarkably better than direct instrumentation (94s vs TO on `gd-5`-WM). For programs where the execution time is not meaningful enough or very close for various techniques, the number of explored paths still clearly illustrates the benefits. Interestingly, DSE$^\star$($P^\star$) does perform better than standard DSE (up to 2x) on a few examples with very few additional paths (e.g. for `get_tag_6` or `gd-6`). We conjecture that additional label constraints may sometimes greatly simplify the solving process by introducing pertinent case considerations, but this point must be investigated further.

From a coverage point of view, DSE$^\star$($P^\star$) performs better than standard DSE on 10/26 examples, with an increase in coverage from 3% up to 39% (`trityp`-MCC). Any increase in coverage coming at a reasonable cost is welcome, since hard-

**Table 2**
Experiments (2/2) for DSE$^\star$ vs DSE and random testing.

| | | | Random<br>(witness) | DSE($P$)<br>(witness) | DSE($P'$) | DSE($P^\star$) | DSE$^\star$($P^\star$) |
|---|---|---|---|---|---|---|---|
| get_tag-5 | CC | #paths | 56,468 | 11,833 | 40,102 | 22,669 | 11,843 |
| 240 loc | 20$l$ | time | 64 s | 60 s | 210 s | 651 s | 64 s |
| | | cover | 20/20 | 20/20 | 20/20 | 20/20 | 20/20 |
| | MCC | #paths | 42,380 | 11,833 | 41,605 | 23,794 | 11,848 |
| | 26$l$ | time | 48 s | 60 s | 100 s | 510 s | 48 s |
| | | cover | 26/26 | 26/26 | 26/26 | 26/26 | 26/26 |
| | WM | #paths | 35,935 | 11,833 | 58,646 | 28,919 | 11,856 |
| | 47$l$ | time | 51 s | 60 s | 140 s | 719 s | 51 s |
| | | cover | 42/47 | 44/47 | 44/47 | 44/47 | 44/47 |
| get_tag-6 | CC | #paths | 1,305,971 | 76,456 | | | 76,468 |
| 240 loc | 20$l$ | time | 1,512 s | 3,011 s | **TO** | TO | **1,512 s** |
| | | cover | 20/20 | 20/20 | | | 20/20 |
| | MCC | #paths | 1,353,672 | 76,456 | | | 76,472 |
| | 26$l$ | time | 1,481 s | 3,011 s | **TO** | TO | **1,481 s** |
| | | cover | 26/26 | 26/26 | | | 26/26 |
| | WM | #paths | 1,287,924 | 76,456 | | | 76,481 |
| | 47$l$ | time | 1,463 s | 3,011 s | **TO** | TO | **1,463 s** |
| | | cover | 44/47 | 44/47 | | | 44/47 |
| gd-5 | CC | #paths | 53,330 | 14,516 | 18,220 | 17,018 | 14,605 |
| 319 loc | 36$l$ | time | 59 s | 51 s | 66 s | 91 s | 59 s |
| | | cover | 16/36 | 36/36 | 36/36 | 36/36 | 36/36 |
| | MCC | #paths | 68,691 | 14,516 | 20,261 | 18,799 | 15,201 |
| | 36$l$ | time | 80 s | 51 s | 71 s | 101 s | 80 s |
| | | cover | 14/36 | 29/36 | 29/36 | 29/36 | 29/36 |
| | WM | #paths | 74,999 | 14,516 | | | 14,607 |
| | 63$l$ | time | 94 s | 51 s | **TO** | TO | **94 s** |
| | | cover | 46/63 | 61/63 | | | 62/63 |
| gd-6 | CC | #paths | 2,785,951 | 107,410 | 131,726 | 125,024 | 107,500 |
| 319 loc | 36$l$ | time | 2,945 s | 3,740 s | 3,816 s | 5,534 s | 2,945 s |
| | | cover | 23/36 | 36/36 | 36/36 | 36/36 | 36/36 |
| | MCC | #paths | 3,247,423 | 107,410 | 144,840 | 137,328 | 111,208 |
| | 36$l$ | time | 3,447 s | 3,740 s | 3,822 s | 6,281 s | 3,447 s |
| | | cover | 18/36 | 29/36 | 29/36 | 29/36 | 29/36 |
| | WM | #paths | 2,064,394 | 107,410 | | | 107,521 |
| | 63$l$ | time | 2,232 s | 3,740 s | **TO** | TO | **2,232 s** |
| | | cover | 52/63 | 62/63 | | | 63/63 |

TO: time-out (5,400 s), x: crash due to a bug in the underlying solver.

**Table 3**
Overhead (slow-down) with respect to DSE.

| | DSE($P'$) | DSE($P^\star$) | DSE$^\star$($P^\star$) |
|---|---|---|---|
| Min | ×1.02 | ×0.92 | ×0.49 |
| Median | ×1.79 | ×2.55 | ×1.37 |
| Max | ×122.50 | ×105.88 | ×7.15 |
| Mean | ×20.29 | ×10.50 | ×2.15 |
| Time-outs* | 5 | 5 | 0 |

* Overheads take into account time-outs, counted as 5400 s (90 min).

**Table 4**
Label coverage ratios.

| | Random | DSE($P$) | DSE($P'$) | DSE($P^\star$) | DSE$^\star$($P^\star$) |
|---|---|---|---|---|---|
| Min | 37% | 61% | 62% | 62% | 62% |
| Median | 63% | 90% | 92% | 93% | 95% |
| Max | 100% | 100% | 100% | 100% | 100% |
| Mean | 70% | 87% | 88% | 88% | 90% |

Time-outs are excluded from the coverage computation.

**Table 5**
Label coverage ratios.

| | Random | DSE($P$) | DSE*($P^\star$) | |
|---|---|---|---|---|
| | | | vanilla | INF |
| Min. | 37% | 61% | 62% | 80% |
| Med. | 63% | 90% | 95% | 100% |
| Max. | 100% | 100% | 100% | 100% |
| Mean | 70% | 87% | 90% | 96% |

For DSE*($P^\star$)-INF, ratios take into account the detected infeasible labels. Time-outs are excluded from the coverage computation.

to-cover test objectives are considered more likely to detect bugs [17]. Recall also that DSE*($P^\star$) offers relative completeness guarantees regarding label coverage, while standard DSE does not.

Finally, we observe that random testing obtains very unstable label coverage (varying between 44% to 100% of the coverable labels, with the average of 70%). Yet, it remains significantly lower than the DSE*($P^\star$) coverage (91% in average).

**Conclusion.** These experiments confirm our formal predictions:

- DSE*($P^\star$) performs dramatically better on difficult programs than the direct instrumentation, both in terms of search space and computation time;
- the overhead of DSE*($P^\star$) w.r.t. standard DSE turns out to be always acceptable and often very low,[8] while label coverage is indeed increased in many cases—sometimes very significantly;
- finally, DSE*($P^\star$) achieves considerably better coverage than random testing for the same time budget.

These results suggest that DSE can be efficiently lifted to **LC** coverage thanks to our optimizations.

### *8.2. Evaluating the impact of infeasible label detection*

**Objectives.** Our framework enables detecting infeasible labels. In this section, we aim at evaluating how efficient this detection can be. We also measure the impact of infeasible label detection over the efficiency of our label-driven test generation. Indeed, if some infeasible labels have been pruned before instrumentation and test generation, DSE*($P^\star$) can be stopped as soon as it has attempted covering all the remaining (presumably feasible) labels, instead of wasting time trying to cover the infeasible labels as well.

**Protocol.** We consider the same annotated benchmark programs and the same three coverage criteria (**CC**, **MCC** and **WM**) as in the previous section.

We compare the following variants of symbolic execution (cf. Tables 6 and 7): the DSE*($P^\star$) technique presented so far[9] and DSE*($P^\star$)-INF that exploits in addition the infeasible labels detected through a preliminary pass with our LUncov module. The computation time of DSE*($P^\star$)-INF does not include the computation time of LUncov, whose results are reported in a separate column. Thanks to the knowledge of at least some infeasible labels, label coverage for DSE*($P^\star$)-INF is computed w.r.t. the set of potentially coverable labels, i.e. labels detected as infeasible are discarded.

**Results.** A total of 84 infeasible labels were identified, spanning over 13/26 programs and topping at 35% of the total number of labels in some examples. This yields a substantial improvement of the reported coverage ratios (see Table 5). In 6 out of 26 cases, the coverage ratios do reach 100% of the *actually feasible* objectives.

When *infeasible objectives* are detected in the tested program, pruning out those infeasible objectives made DSE* in average 2.18x faster, the speedup topping at 11.66x. Results on the whole benchmark are more mitigated (cf. Tables 6 and 7). Yet, note that the detection of infeasible labels takes an acceptable amount of time w.r.t. the test generation time on our benchmark programs (12% of the total computation time in average), and induces almost no slow-down for the bigger examples (less than 3% when test generation takes more than 10 s).

**Conclusion.** The experiments confirm the ability of our framework to detect infeasible labels efficiently and the interest of doing so to speed up test generation.

[8] The overhead is even further reduced with the optimizations presented in Section 8.2.

[9] With the additional ability to stop when all the labels have been covered, instead of terminating only when all the DSE-reachable paths within a chosen upper bound for the total number of loop iterations have been explored.

**Table 6**
Experiments (1/2) for DSE⋆ with infeasible label detection.

| | | | Random<br>(witness) | DSE($P$)<br>(witness) | DSE⋆($P^\star$) | Uncov.<br>detect. | DSE⋆($P^\star$)<br>INF |
|---|---|---|---|---|---|---|---|
| trityp<br>50 loc | CC<br>24 $l$ | #paths | 1,500 | 35 | 35 | | 35 |
| | | time | 1.6 s | 1.3 s | 1.4 s | 0.6 s | 1.4 s |
| | | cover | 12/24 | 24/24 | 24/24 | 0/24 | 24/24 |
| | MCC<br>28 $l$ | #paths | 1,619 | 35 | 51 | | 51 |
| | | time | 2.1 s | 1.3 s | 1.9 s | 0.5 s | 1.9 s |
| | | cover | 16/28 | 17/28 | 28/28 | 0/28 | 28/28 |
| | WM<br>129 $l$ | #paths | 4,301 | 35 | 98 | | 83 |
| | | time | 5.1 s | 1.3 s | 5.0 s | 0.7 s | 5.0 s |
| | | cover | 61/129 | 112/129 | 125/129 | 4/129 | 125/125 |
| 4balls<br>35 loc | WM<br>67 $l$ | #paths | 1,903 | 7 | 23 | | 23 |
| | | time | 2.1 s | 1.2 s | 2.1 s | 0.5 s | 2.1 s |
| | | cover | 25/67 | 55/67 | 56/67 | 56/67 | 56/67 |
| utf8-3<br>108 loc | WM<br>84 $l$ | #paths | 2,551 | 134 | 313 | | 283 |
| | | time | 3.8 s | 1.4 s | 3.8 s | 0.5 s | 2.7 s |
| | | cover | 52/84 | 53/84 | 55/84 | 29/84 | 55/55 |
| utf8-5<br>108 loc | WM<br>84 $l$ | #paths | 5,396 | 680 | 743 | | 189 |
| | | time | 8.1 s | 2.0 s | 8.1 s | 0.6 s | 3.5 s |
| | | cover | 53/84 | 80/84 | 82/84 | 2/84 | 82/82 |
| utf8-7<br>108 loc | WM<br>84 $l$ | #paths | 23,053 | 3,069 | 3,265 | | 152 |
| | | time | 35 s | **5.8 s** | 35 s | 0.6 s | **3.0 s** |
| | | cover | 53/84 | 80/84 | 82/84 | 2/84 | 82/82 |
| tcas | CC<br>10 $l$ | #paths | 3,096 | 2,787 | 255 | | 255 |
| | | time | 3.4 s | 2.9 s | 1.6 s | 0.5 s | 1.6 s |
| | | cover | 10/10 | 10/10 | 10/10 | 10/10 | 10/10 |
| | MCC<br>12 $l$ | #paths | 3,182 | 2,787 | 3,059 | | 3,059 |
| | | time | 3.9 s | 2.9 s | 3.9 s | 0.5 s | 3.9 s |
| | | cover | 11/12 | 10/12 | 11/12 | 0/12 | 11/12 |
| tcas' | WM<br>111 $l$ | #paths | 20,347 | 4,420 | 6,014 | | 5,388 |
| | | time | 27 s | 5.6 s | 27 s | 0.7 s | 26 s |
| | | cover | 90/111 | 88/111 | 101/111 | 6/111 | 101/105 |
| replace<br>100 loc | WM<br>79 $l$ | #paths | 11,747 | 866 | 2,347 | | 126 |
| | | time | 14 s | 2 s | 14 s | 0.8 s | 13 s |
| | | cover | 68/79 | 69/79 | 70/79 | 5/79 | 70/74 |
| full_bad<br>219 loc | CC<br>16 $l$ | #paths | 5,180 | 2,563 | 3,209 | | 2,648 |
| | | time | 7 s | 5 s | 7 s | 0.5 s | 6.5 s |
| | | cover | 9/16 | 12/16 | 12/16 | 2/16 | 12/14 |
| | MCC<br>39 $l$ | #paths | 9,393 | 2,563 | 7,043 | | 3,470 |
| | | time | 19 s | 5 s | 19 s | 0.6 s | 13 s |
| | | cover | 17/39 | 24/39 | 24/39 | 9/39 | 24/30 |
| | WM<br>46 $l$ | #paths | 9,425 | 2,563 | 5,414 | | 2,976 |
| | | time | 19 s | 5 s | 19 s | 0.7 s | 14 s |
| | | cover | 29/46 | 34/46 | 34/46 | 7/46 | 34/39 |

loc: number of lines of code, $l$: number of labels.

### 8.3. Experiments with a complex coverage criterion

**Objectives.** We want to check if the results of Sections 8.1 and 8.2 still hold for **GACC**, which is a more complex criterion, known as challenging for test automation tools [25].

**Protocol and results.** We reuse the same protocols and benchmark programs that those in Sections 8.1 and 8.2. Yet, we restrict ourselves to the 6 programs with decision predicates involving more than one atomic condition. Results are reported in Tables 8, 9, and 10. DSE⋆($P^\star$) achieves good coverage results (>90% on 4/6 examples, average 85%), significantly better than random testing and DSE. Yet, the overhead is significantly more important than for the other criteria (max. of 60x vs 7x, average of 10.55x vs 2.15x). Compared with direct instrumentation, DSE⋆($P^\star$) does avoid a time-out and is twice as fast on average (excluding TO). Finally, infeasible label detection allows to recover a reasonable overhead (max 3.14x, average 1.85x) and to detect 37 uncoverable labels out of 91 uncovered ones.

**Conclusion.** Experiments confirm that **GACC** is a challenging criterion for automatic tools. Yet, while the overhead of DSE⋆($P^\star$) is more important than it is with other criteria, it is still affordable in most cases, and the technique achieves very good coverage—much better than random testing or standard DSE. Infeasible label detection keeps overhead low. Surprisingly, **MCC** seems significantly easier to cover on these examples than **GACC**, while it is more powerful. We conjecture

**Table 7**
Experiments (2/2) for DSE$^\star$ with infeasible label detection.

| | | | Random<br>(witness) | DSE($P$)<br>(witness) | DSE$^\star$($P^\star$) | Uncov.<br>detect. | DSE$^\star$($P^\star$)<br>INF |
|---|---|---|---|---|---|---|---|
| get_tag-5 | CC | #paths | 56,468 | 11,833 | 94 | | 94 |
| 240 loc | 20 $l$ | time | 64 s | **60 s** | **2.1 s** | 0.7 s | **2.1 s** |
| | | cover | 20/20 | 20/20 | 20/20 | 0/20 | 20/20 |
| | MCC | #paths | 42,380 | 11,833 | 98 | | 98 |
| | 26 $l$ | time | 48 s | **60 s** | **1.4 s** | 0.7 s | **1.4 s** |
| | | cover | 26/26 | 26/26 | 26/26 | 0/26 | 26/26 |
| | WM | #paths | 35,935 | 11,833 | 11,856 | | 11,856 |
| | 47 $l$ | time | 51 s | 60 s | 51 s | 0.7 s | 53 s |
| | | cover | 42/47 | 44/47 | 44/47 | 2/47 | 44/45 |
| get_tag-6 | CC | #paths | 1,305,971 | 76,456 | 306 | | 306 |
| 240 loc | 20 $l$ | time | 1,512 s | **3,011 s** | **2.6 s** | 0.6 s | **2.6 s** |
| | | cover | 20/20 | 20/20 | 20/20 | 0/20 | 20/20 |
| | MCC | #paths | 1,353,672 | 76,456 | 310 | | 310 |
| | 26 $l$ | time | 1,481 s | **3,011 s** | **2.6 s** | 0.6 s | **2.6 s** |
| | | cover | 26/26 | 26/26 | 26/26 | 0/26 | 26/26 |
| | WM | #paths | 1,287,924 | 76,456 | 76,481 | | 76,481 |
| | 47 $l$ | time | 1,463 s | 3,011 s | 1,463 s | 0.7 s | 1,281 s |
| | | cover | 44/47 | 44/47 | 44/47 | 2/47 | 44/45 |
| gd-5 | CC | #paths | 53,330 | 14,516 | 10,386 | | 10,386 |
| 319 loc | 36 $l$ | time | 59 s | 51 s | 34 s | 1.2 s | 34 s |
| | | cover | 16/36 | 36/36 | 36/36 | 0/36 | 36/36 |
| | MCC | #paths | 68,691 | 14,516 | 15,201 | | 10,443 |
| | 36 $l$ | time | 80 s | 51 s | 80 s | 1.9 s | **47 s** |
| | | cover | 14/36 | 29/36 | 29/36 | 7/36 | 29/29 |
| | WM | #paths | 74,999 | 14,516 | 14,607 | | 14,607 |
| | 63 $l$ | time | 94 s | 51 s | 94 s | 1.7 s | 94 s |
| | | cover | 46/63 | 61/63 | 62/63 | 0/63 | 62/63 |
| gd-6 | CC | #paths | 2,785,951 | 107,410 | 77,780 | | 77,780 |
| 319 loc | 36 $l$ | time | 2,945 s | **3,740 s** | **1,577 s** | 1.1 s | **1,577 s** |
| | | cover | 23/36 | 36/36 | 36/36 | 0/36 | 36/36 |
| | MCC | #paths | 3,247,423 | 107,410 | 111,208 | | 77,851 |
| | 36 $l$ | time | 3,447 s | **3,740 s** | 3,447 s | 1.9 s | **1,557 s** |
| | | cover | 18/36 | 29/36 | 29/36 | 7/36 | 29/29 |
| | WM | #paths | 2,064,394 | 107,410 | 77,796 | | 77,796 |
| | 63 $l$ | time | 2,232 s | **3,740 s** | **1,445 s** | 1.5 s | **1,445 s** |
| | | cover | 52/63 | 62/63 | 63/63 | 0/63 | 63/63 |

loc: number of lines of code, $l$: number of labels.

that our **GACC** encoding may lead to too complex formula to solve. Though, this encoding is very succinct (linear) in the number of atomic conditions while the **MCC** encoding is exponential, which could make a huge difference on very complex branching conditions.

Overall, taking into account all the coverage criteria considered in Section 8.1 plus **GACC**, overhead is kept small (no time-out, average 3.93x for DSE$^\star$($P^\star$), 1.57x with infeasible label detection), and reported coverage ratio is very high (average of 90%, median value 94%)—especially with the help of infeasibility detection (average 95%, median value 100%).

## 9. Threats to validity

The validity of our results has been crosschecked in several ways:

- We verify that all the labels covered by DSE($P$) are also consistently covered by DSE($P^\star$) and that all the labels covered by DSE($P^\star$) are also consistently covered by DSE$^\star$($P^\star$) (both properties are satisfied as all DSE variants are run in the same deterministic way, so that the optimizations of more advanced variants free up additional test budget to cover more labels).
- We replay the test suites generated by all DSE variants using the LReplay tool and check that the reported coverage ratios are consistent with those reported by DSE($P^\star$) and DSE$^\star$($P^\star$).
- We check that all the labels covered by any technique (including random testing) are disjoint from the labels identified as infeasible by our approach.
- We specifically test that DSE$^\star$($P^\star$) does not try to branch on an already covered label and that all path constraints from DSE($P^\star$) have at most one label constraint (by reviewing a fraction of the output logs for constraint solving and covered labels).

**Table 8**
Results for **GACC**.

| | | Random | DSE($P$) | DSE($P'$) | DSE$^\star$($P^\star$) | Uncov. | DSE$^\star$($P^\star$) |
|---|---|---|---|---|---|---|---|
| | | (witness) | (witness) | | | detect. | INF |
| trityp | #paths | 1,438 | 56 | 65 | 61 | | 56 |
| 34$l$ | time | 2.3 s | 1.3 s | 2.3 s | 2.3 s | 0.7 s | 1.8 s |
| | cover | 16/34 | 34/34 | 34/34 | 34/34 | 0/34 | 34/34 |
| tcas | #paths | 6,319 | 2,787 | 10,168 | 4,234 | | 4,201 |
| 46$l$ | time | 9.1 s | 2.9 s | 10.9 s | 9.1 s | 0.7 s | 9.1 s |
| | cover | 46/46 | 46/46 | 46/46 | 46/46 | 0/46 | 46/46 |
| full_bad | #paths | 17,562 | 2,563 | 10,000 | 3,619 | | 3,041 |
| 34$l$ | time | 23 s | 5.0 s | 27 s | 23 s | 3.7 s | 11.6 s |
| | cover | 22/39 | 24/39 | 30/34 | 30/34 | 2/34 | 30/32 |
| get_tag-5 | #paths | 33,877 | 7,456 | 116,121 | 8,077 | | 7,742 |
| 94$l$ | time | 73 s | 40 s | 330 s | 73 s | 1.2 s | 58 s |
| | cover | 59/94 | 58/94 | 60/94 | 60/94 | 11/94 | 60/83 |
| get_tag-6 | #paths | 1,123,829 | 47,216 | 753,409 | 50,957 | | 48,819 |
| 94$l$ | time | 2,388 s | 1,467 s | 5,258 s | 2,388 s | 1.5 s | 2,274 s |
| | cover | 60/94 | 59/94 | 61/94 | 61/94 | 10/94 | 61/84 |
| gd-5 | #paths | 45,876 | 14,516 | 52,419 | 15,459 | | 15,401 |
| 76$l$ | time | 90 s | 51 s | 134 s | 90 s | 7.9 s | 92 s |
| | cover | 40/76 | 58/76 | 66/76 | 66/76 | 7/76 | 66/69 |
| gd-6 | #paths | 3,723,361 | 107,410 | | 113,044 | | 112,671 |
| 76$l$ | time | 5,269 s | 3,740 s | TO | 5,269 s | 7.9 s | 5,241 s |
| | cover | 48/76 | 58/76 | | 66/76 | 7/76 | 66/69 |

**Table 9**
Overhead (slow-down) w.r.t. DSE for GACC.

| | DSE($P'$) | DSE$^\star$($P^\star$) | |
|---|---|---|---|
| | | vanilla | INF |
| 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× |
| Time-outs* | 1 | 0 | 0 |

* For DSE($P'$), a time-out counts as a 5400 s (90 min).

**Table 10**
Label coverage ratios for **GACC**.

| | Random | DSE($P$) | DSE$^\star$($P^\star$) | |
|---|---|---|---|---|
| | | | vanilla | INF |
| Min | 47% | 62% | 64% | 72% |
| Med | 55% | 76% | 88% | 96% |
| Max | 100% | 100% | 100% | 100% |
| Mean | 60% | 78% | 85% | 91% |

For DSE$^\star$($P^\star$)-INF, ratios take into account the detected infeasible labels. Time-outs are excluded from the coverage computation.

Another class of threats may arise because of the tool implementations we used, as it cannot be completely excluded that Frama-C or our implementation are defective. However, Frama-C is a mature tool with industrial applications in highly demanding fields (e.g., aeronautics) and thus, it is unlikely to cause important problems. Moreover, our sanity checks would have likely spotted such issues.

While we have not directly evaluated the bug-finding power of label-based DSE, we have relied on common coverage criteria, whose bug-finding capabilities have already been extensively studied.

Common to all studies relying on empirical data, the present study may be of limited generalizability. To diminish this threat we used 12 standard benchmark programs, mainly coming from the Siemens test suite, the Verisec benchmark and MediaBench, and explored overall more than 10 millions of execution paths. While dynamic symbolic execution might still face scalability challenges in some situations, due to path explosion and constraint solving, it is yet a popular and powerful test generation technique, successfully applied in many industrial contexts. Additionally, our label framework is not tied to symbolic execution but could also be integrated with other test generation approaches. As demonstrated in [19], detecting infeasible labels can scale up to large programs, via a compositional and parallel applications of assertion checkers.

Our results might also have been affected by the choice of the considered coverage criteria and in particular the specific mutation operators we employ. To reduce this threat, we used popular coverage criteria (CC, MCC, GACC and WM) included in software testing standards [13,12], and employed commonly used mutation operators included in recent work [11,10].

Finally, other threats may be due to the form of the infeasible objectives that we target, as well as the significance of their number in the considered software. However, infeasible objectives are a well-known issue, usually acknowledged in the literature as one of the main cost factors of the software testing process [16,15,14]. To reduce this threat, we considered several coverage criteria and various examples of programs in the benchmarks.

## 10. Extensions and applications

We summarize here some recent extensions to the framework for automated test generation presented in this paper.

### *10.1. Extending the expressive power of labels: hyperlabels*

While labels can express a large range of criteria (including a large part of weak mutations **WM**, and the weak **GACC** variant of **MCDC**), they still face some limits in terms of expressiveness. For instance, labels cannot express strong variants of **MCDC**, higher-order mutations or most path and dataflow criteria directly. In [8,9], we introduce five simple label combination or enrichment operators: value bindings, sequences, guards, conjunctions and disjunctions. By combining and enriching labels using these operators, one can build new and more complex objectives to be covered, called hyperlabels. Hyperlabels are able to encode all criteria from the literature but full mutations, and enable specifying test objectives aiming at detecting violations of complex security properties, such as non-interference. Lifting our framework to provide efficient test generation and infeasible objective detection for hyperlabels is interesting future work.

### *10.2. Pruning out redundant and infeasible labels, at scale*

As full objective coverage is rarely reached in practice, testers rely on the ratio of covered objectives to measure the strength of their test suites. However, the working assumption of this practice is that all objectives are of equal value. Testing research demonstrated that this is not true, as duplication and subsumption can make a large number of feasible test objectives redundant. Such feasible redundant objectives may artificially deflate or inflate the coverage ratio, skewing the measurement, which may misestimate test thoroughness and fail to evaluate correctly the remaining cost to full coverage. The approach introduced in [19] is similar to the one used here to detect infeasible labels and enables one to prune out many redundant pairs of labels. The proof of redundancy for a pair of labels is reduced to a proof of validity for a corresponding assertion in the code, which is delegated to an off-the-shelf assertion prover.

Infeasible and redundant label detection relies on using an assertion prover as a back-end, whose complex analyses may poorly scale to real world programs. Marcozzi et al. [19] also demonstrate that scalable detection of infeasible and redundant labels is possible, using local-scoped compositional analyses and a multicore implementation. Experiments over large real-world applications such as OpenSSL and SQLite show that one can process hundreds of thousands of labels in hundreds of thousands of lines of code in acceptable time, pruning out more than 10% of the labels as either infeasible or redundant.

### *10.3. Ongoing industrial adoption of label-based technology*

Recent work [33] reports on an ongoing industrial adoption and further enhancements of label-based testing tools at MERCE, a research center of Mitsubishi Electric. MERCE performed experiments with an industrially adapted and extended version of LTest (including program annotation, detection of infeasible objectives and test generation) on a real industrial code (with about 1,300 functions and 80,000 lines of C code). The labels were generated automatically thanks to the program annotation module. The detection of infeasible objectives using static analysis was very efficient and classified 8.3% of objectives as infeasible, within a couple of minutes. Test generation for the remaining labels covered 86% of test objectives and took about 8 hours. MERCE roughly estimated the effective time benefit factor compared to the manual testing as more than 230x for test input generation.

An even more recent experiment combining label-based testing with genetic search-based test-generation techniques reported by MERCE [34] lead to the *classification* (that is, either covering by a test case or proving to be infeasible) of more than 99% of test objectives out of 20,000 objectives on a real industrial code (with about 82,000 lines of C code). Those very good results are very encouraging for further pushing the technology in industry.

### *10.4. Using labels for combinations of testing and proof*

An interesting usage of label coverage appeared in recent research on combinations of testing and proof. In the context of certification of avionic software [12] (e.g. according the DO-178C norm, level A), the verification engineer must demonstrate sufficient testing coverage of both code (structural coverage) and specification (functional coverage). When some parts of code are formally proved, while others are tested, the requirements on the coverage should be modified. Recent work [35] proposed a new notion of verification coverage based on the notion of labels and mutation coverage. It also proposed a methodology to ensure that a verification campaign is complete with respect to this coverage. It allows the verification engineer to combine testing tools and provers in the verification process and to reduce the verification cost.

### *10.5. Using labels for detecting polluting test objectives for data-flow criteria*

For dataflow criteria, test objectives are often expressed using the notion of a *def-use pair*, linking a statement where a variable is defined (written) to a statement where it is used (read) without being redefined in between. A def-use pair can be defined as a sequence of two labels with an additional condition on the path between them (that must be *def-clear* for that variable). Def-use pairs are combined to form more complex criteria such as all-defs and all-uses. They are examples of criteria expressible in hyperlabels (cf. Section 10.1).

Recent work [1] extended the LUncov module of LTest to the detection of polluting test objectives for dataflow criteria using various static analysis techniques: dataflow analysis, value analysis and weakest precondition calculus. The reported experiments (on programs of up to 11000 loc) show that 64% of objectives were identified as polluting (non-inapplicable, infeasible or redundant). The analysis time remained acceptable (taking at most 64 min. on the longest example with ∼45000 test objectives).

## 11. Related work

**Lifting DSE to various coverage criteria.** The need for enhancing DSE with better coverage criteria has already been pointed out in active testing (also known as assertion-based testing) [40,47,52], Mutation DSE [64,65] and Augmented DSE [50,67,76]. The present work generalizes these results and proposes ways of taming the potential blow-up, resulting in an effective support of advanced coverage criteria in DSE with only a small overhead. More precisely, we give a more generic view of the problem, identifying labels and annotated programs as the key concept underlying the approach. We also clearly state the limits and hypotheses of the method by introducing the notions of simulation and labeling functions, identifying the side-effect free fragment of **WM**, proving soundness of direct instrumentation and providing a formalization of the path space complexification induced by direct instrumentation. Most importantly, we propose the tight instrumentation which is proved to completely prevent complexification. Finally, our optimizations can be implemented in a pure black-box setting and we do not impose any specific search heuristics, keeping room for future improvements.

Active testing targets run-time errors by adding explicit branches into the program. It is similar to the Run-Time Error Coverage criterion presented in Section 4. Labels are a more general approach. The direct instrumentation $P'$ for this criterion is mostly equivalent to $P^\star$ since additional branches can only trigger errors and stop the execution. Yet, active testing could benefit from the IDL optimization. Finally, since most test objectives are (hopefully!) uncoverable for Run-Time Error Coverage, some approaches aim at combining DSE with static detection of uncoverable targets [39,40]. They can be reused for labels, and should be useful when many labels are uncoverable.

Following Offut et al. [44], Papadakis et al. show that **WM** can be reduced to branch coverage through the use of a variant of Mutant Schemata [71]. This is pretty similar to the direct encoding $P'$ mentioned here. They propose essentially two variations of DSE for mutation testing: a black-box approach [64] based on a direct encoding similar to our DSE($P'$) scheme, and a more ad hoc approach [64] preventing reuse of existing DSE tools but offering several optimizations. Papadakis et al. propose a variant of IDL, a dedicated search heuristic based on shortest paths [63] and an improvement of the direct encoding through the use of mutant identifiers (following exactly Mutant Schemata). On the one hand, it ensures that a given path cannot go through several *different* mutants, on the other hand there is still an exponential blow-up of the search space in the worst case, and IDL cannot cover more than one mutant at once.

Augmented DSE [50] is a variant of direct instrumentation. Several coverage criteria are encoded, getting results similar to those of Section 4.2, yet the side-effect free subset of **WM** is not identified. Experiments [50, Table 2] report an average time-overhead of 272x, going up to >2,000x. That confirms the strong benefits of our optimizations, that yield a maximal overhead of 7x (60x with **GACC**). Following the same approach, Pandita et al. show that **GACC** can be simulated through direct instrumentation [67]. This result can be directly recast in terms of labels, cf. Section 8.3.

In the same line of ideas, JPF allows specifying complex coverage criteria through temporal logic formulas [77]. This is slightly more expressive than labels because temporal formulas can express constrained reachability objectives (e.g. dataflow criteria) while labels are in principle limited to strict reachability. Yet, constrained reachability can be reduced to strict reachability with proper instrumentation. Labels describe test objectives in a somewhat less convenient way, yet we propose an in-depth analysis of their expressive power in terms of coverage criteria (especially mutations) and a very efficient support in symbolic execution. Note that JPF provides an encoding of Masking **MCDC** (a.k.a. **CACC** [25]), a criterion standing in between **GACC** and **MCDC**. Their approach [77] can be recast in terms of labels.

**Labels and optimized DSE.** The label-specific optimizations described here can be freely mixed with other DSE optimizations. It is left as future work to explore which optimizations turn out to be the most effective for labels. As already stated, combining static discovery of uncoverable labels with DSE [39,40] could be useful for often-uncoverable labels, such as those generated for Run-Time Error Coverage or **MCC**. First results in that direction [26] are presented in Section 8.2, more recent results about uncoverability detection can be found in [27]. Other promising directions are to adapt DSE search heuristics [74] by taking advantage of the dissimilarities between labels and branches, possibly getting inspiration from [63] and [69], or to distribute the DSE search thanks to a static pre-partitioning of test objectives [78].

The IDL optimization shows some similarities with Look-Ahead pruning (LA) [29,36]. Basically, LA takes advantage of (global) static analysis to prune partial paths which cannot reach any uncovered branches. In particular, on $P^\star$, IDL prunes several label-terminated paths at once thanks to dynamic analysis, which is orthogonal to LA.

**Automation of mutation testing.** Mutation coverage [43,60] has been established as a powerful criterion through several experimental studies [24,60]. Yet, it is very difficult to automate. Even mutation score computation is expensive in practice if not done wisely. Weak mutations [49] relax mutation coverage by abandoning the "propagation step", making **WM** easier to compare with standard criteria and easier to test for. **WM** has been experimentally proved to be almost equivalent to strong mutations [58], and from a theoretical point of view **WM** subsumes many other criteria [61].

The few existing symbolic methods for mutation-based ATG rely on the encoding proposed by Offutt et al. and have already been discussed [44,66,65]. The Mutation Schemata technique [71] was originally developed in order to factorize the compilation costs of hundreds of similar mutants. Static analysis has been proposed for the "equivalent mutant detection" problem [56,55] in a way similar to what is sketched in Section 7.

The side-effect free fragment of **WM** presented in this paper seems to be a sweet spot of mutation testing: it is amenable to efficient automation and still very expressive. It is left as future work to identify if something essential is lost within this fragment. Finally, our encoding of **WM** into **LC** is orthogonal to and can be combined with some of the many techniques developed for efficient mutation testing, such as operator reduction [59,72] or smart use of operators [51].

**Property-based testing.** Property-based testing [5] uses an automatic test generation tool to find violations of some semantic properties by the program under test. Contrary to labels, which are structural test objectives generated automatically and massively from a syntactic analysis of the program under test, property-based testing targets violations of a limited number of hand-written partial specifications of the program's semantics. While some recent works have advocated the use of more directed test generation approaches in property-based testing [6,7], state-of-the-art tools rely on random test generation to find property violations and do not take advantage of static analyses to prove correct properties.

## 12. Conclusion

In this paper, we present a complete panorama of different achievements related to the label coverage criterion. Label coverage appears to be both expressive and amenable to efficient automation. Some of the ideas behind labels underlie previous work by other teams. We generalize them, propose ways of taming the potential complexification of the path space and provide both formal and experimental evidence. Especially, we have shown how to extend DSE for label coverage in a black-box manner with an acceptable overhead and how to efficiently detect infeasible test objectives by existing assertion checkers. Experiments show that our optimizations yield very significant improvements. Recent applications and ongoing adoption of the label coverage based techniques in industry [34,33] demonstrate the practical interest of the technology.

This work also bridges part of the gap between symbolic ATG techniques and coverage criteria. On the one hand, we show that DSE techniques can be cheaply extended to support more advanced coverage criteria, including side-effect free weak mutations. On the other hand, we identify a powerful criterion amenable to efficient automation.

As a whole, label coverage forms the basis of a very generic and convenient framework for test automation, providing a powerful specification mechanism for test objectives and featuring efficient integration into symbolic ATG techniques.

## CRediT authorship contribution statement

**Sébastien Bardin:** Conceptualization, Formal analysis, Investigation, Methodology, Resources, Validation, Writing – original draft, Writing – review & editing. **Nikolai Kosmatov:** Conceptualization, Formal analysis, Investigation, Methodology, Resources, Validation, Writing – original draft, Writing – review & editing. **Michaël Marcozzi:** Formal analysis, Investigation, Methodology, Resources, Validation, Writing – original draft, Writing – review & editing. **Mickaël Delahaye:** Formal analysis, Investigation, Methodology, Resources, Validation, Writing – original draft.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Acknowledgement

The authors would like to thank the Frama-C and PathCrawler team members for providing the tools and support, François Cheynier for initial experiments with labels, as well as David R. MacIver and Alastair F. Donaldson for useful discussions about property-based testing. Many thanks to the anonymous reviewers for many helpful comments and suggestions. This work was partially funded by EU FP7 (project STANCE, grant 317753) and French ANR (project BINSEC, grant ANR-12-INSE-0002).

## References

[1] T. Martin, N. Kosmatov, V. Prevosto, M. Lemerre, Detection of polluting test objectives for dataflow criteria, in: 16th International Conference on Integrated Formal Methods, IFM, Springer, 2020, pp. 337–345.

[2] P.S. Kochhar, F. Thung, D. Lo, Code coverage and test suite effectiveness: empirical study with real bugs in large systems, in: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering, SANER, 2015, pp. 560–564.

[3] R. Just, D. Jalali, L. Inozemtseva, M.D. Ernst, R. Holmes, G. Fraser, Are mutants a valid substitute for real faults in software testing?, in: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014, Association for Computing Machinery, New York, NY, USA, 2014, pp. 654–665.

[4] C. Cadar, V. Ganesh, P.M. Pawlowski, D.L. Dill, D.R. Engler, Exe: automatically generating inputs of death, ACM Trans. Inf. Syst. Secur. 12 (2) (Dec. 2008), https://doi.org/10.1145/1455518.1455522.

[5] K. Claessen, J. Hughes, Quickcheck: a lightweight tool for random testing of Haskell programs, SIGPLAN Not. 46 (4) (2011) 53–64, https://doi.org/10.1145/1988042.1988046.

[6] L. Lampropoulos, M. Hicks, B.C. Pierce, Coverage guided, property based testing, Proc. ACM Program. Lang. 3 (OOPSLA) (Oct. 2019), https://doi.org/10.1145/3360607.

[7] L. Bulwahn, The new quickcheck for Isabelle, in: C. Hawblitzel, D. Miller (Eds.), Certified Programs and Proofs, Springer, Berlin, Heidelberg, 2012, pp. 92–108.

[8] Michaël Marcozzi, Mickaël Delahaye, Sébastien Bardin, Nikolai Kosmatov, Virgile Prevosto, Generic and effective specification of structural test objectives, in: ICST'17, IEEE, 2017.

[9] Michaël Marcozzi, Sébastien Bardin, Mickaël Delahaye, Nikolai Kosmatov, Virgile Prevosto, Taming coverage criteria heterogeneity with LTest, in: ICST'17, IEEE, 2017.

[10] Thierry Titcheu Chekam, Mike Papadakis, Yves Le Traon, Mark Harman, An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption, in: ICSE 2017, IEEE, 2017.

[11] Paul Ammann, Márcio Eduardo Delamaro, Jeff Offutt, Establishing theoretical minimal sets of mutants, in: ICST 2014, IEEE, 2014.

[12] Radio Technical Commission for Aeronautics, RTCA DO178-B Software Considerations in Airborne Systems and Equipment Certification, 1992.

[13] Stuart C. Reid, The software testing standard — how you can use it, in: EuroSTAR'95, 1995.

[14] D. Yates, N. Malevris, Reducing the effects of infeasible paths in branch testing, in: Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification, TAV3, ACM, New York, NY, USA, 1989, pp. 48–54.

[15] M.R. Woodward, D. Hedley, M.A. Hennell, Experience with path analysis and testing of programs, IEEE Trans. Softw. Eng. 6 (3) (May 1980) 278–286.

[16] E.J. Weyuker, More experience with data flow testing, IEEE Trans. Softw. Eng. 19 (9) (Sept. 1993) 912–919.

[17] P.G. Frankl, O. Iakounenko, Further empirical studies of test effectiveness, Softw. Eng. Notes 23 (6) (1998) 153–162.

[18] T. Su, Z. Fu, G. Pu, J. He, Z. Su, Combining symbolic execution and model checking for data flow testing, in: IEEE/ACM 37th IEEE International Conference on Software Engineering, 2015.

[19] Michaël Marcozzi, Sébastien Bardin, Nikolai Kosmatov, Mike Papadakis, Virgile Prevosto, Loïc Correnson, Time to clean your test objectives, in: 40th International Conference on Software Engineering, ICSE 2018, 2018.

[20] Paul Dan Marinescu, Cristian Cadar, KATCH: high-coverage testing of software patches, in: European Software Engineering Conference / ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2013, 2013.

[21] C. Cadar, V. Ganesh, P.M. Pawlowski, D.L. Dill, D.R. Engler, EXE: automatically generating inputs of death, ACM Trans. Inf. Syst. Secur. 12 (2) (2008) 10.

[22] Leonardo De Moura, Nikolaj Bjørner, Satisfiability modulo theories: introduction and applications, Commun. ACM 54 (9) (2011) 69–77.

[23] Cristian Cadar, Koushik Sen, Symbolic execution for software testing: three decades later, Commun. ACM (February 2013).

[24] J.H. Andrews, L.C. Briand, Y. Labiche, Is mutation an appropriate tool for testing experiments?, in: ICSE 2005, IEEE, 2005.

[25] P. Ammann, A.J. Offutt, Introduction to Software Testing, Cambridge University Press, New York, 2008.

[26] S. Bardin, O. Chebaro, M. Delahaye, N. Kosmatov, An all-in-one toolkit for automated white-box testing, in: TAP 2014, Springer, Heidelberg, 2014.

[27] 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, 2015.

[28] B. Botella, M. Delahaye, S. Hong-Tuan-Ha, N. Kosmatov, P. Mouy, M. Roger, N. Williams, Automating structural testing of C programs: experience with PathCrawler, in: AST 2009, IEEE, 2009.

[29] S. Bardin, P. Herrmann, Pruning the search space in path-based test generation, in: ICST 2009, IEEE, 2009.

[30] S. Bardin, P. Herrmann, Structural testing of executables, in: IEEE ICST 2008, IEEE, 2008.

[31] S. Bardin, P. Herrmann, OSMOSE: automatic structural testing of executables, Softw. Test. Verif. Reliab. 21 (1) (2011) 29–54.

[32] S. Bardin, N. Kosmatov, F. Cheynier, Efficient leveraging of symbolic execution to advanced coverage criteria, in: ICST 2014, IEEE, 2014.

[33] S. Bardin, N. Kosmatov, B. Marre, D. Mentré, N. Williams, Test case generation with PathCrawler/LTest: how to automate an industrial testing process, in: ISOLA 2018, Springer, 2018.

[34] E. Lavillonnière, D. Mentré, D. Cousineau, Fast, automatic, and nearly complete structural unit-test generation combining genetic algorithms and formal methods, in: TAP 2019, Springer, 2019.

[35] V. Le Hoang, L. Correnson, J. Signoles, V. Wiels, Verification coverage for combining test and proof, in: TAP 2018, Springer, 2018.

[36] J. Burnim, K. Sen, Heuristics for scalable dynamic test generation, in: ASE 2008, IEEE, 2008.

[37] 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.

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

[39] O. Chebaro, N. Kosmatov, A. Giorgetti, J. Julliand, Combining static analysis and test generation for C program debugging, in: TAP 2010, Springer, 2010.

[40] O. Chebaro, N. Kosmatov, A. Giorgetti, J. Julliand, Program slicing enhances a verification technique combining static and dynamic analysis, in: SAC 2012, ACM, 2012.

[41] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, B. Yakobowski, Frama-C - a software analysis perspective, in: SEFM 2012, Springer, 2012.

[42] J.J. Chilenski, S.P. Miller, Applicability of modified condition/decision coverage to software testing, Softw. Eng. J. 9 (5) (1994) 193–200.

[43] R.A. DeMillo, R.J. Lipton, A.J. Perlis, Hints on test data selection: help for the practicing programmer, Computer 11 (4) (1978) 34–41.

[44] R.A. DeMillo, A.J. Offutt, Constraint-based automatic test data generation, IEEE Trans. Softw. Eng. 17 (9) (1991).

[45] P. Godefroid, N. Klarlund, K. Sen, DART: directed automated random testing, in: PLDI 2005, ACM, 2005.

[46] P. Godefroid, M.Y. Levin, D. Molnar, Automated whitebox fuzz testing, in: NDSS, 2008.

[47] P. Godefroid, M.Y. Levin, D. Molnar, Active property checking, in: EMSOFT 2008, ACM, 2008.
[48] P. Godefroid, M.Y. Levin, D.A. Molnar, SAGE: whitebox fuzzing for security testing, Commun. ACM 55 (3) (2012) 40–44.
[49] W.E. Howden, Weak mutation testing and completeness of test sets, IEEE Trans. Softw. Eng. 8 (4) (1982).
[50] K. Jamrozik, G. Fraser, N. Tillmann, J. de Halleux, Generating test suites with augmented dynamic symbolic execution, in: TAP 2013, Springer, 2013.
[51] R. Just, G.M. Kapfhammer, F. Schweiggert, Do redundant mutants affect the effectiveness and efficiency of mutation analysis?, in: ICST 2012, IEEE, 2012.
[52] B. Korel, A.M. Al-Yami, Assertion-oriented automated test data generation, in: ICSE 1996, IEEE, 1996.
[53] J.C. King, Symbolic execution and program testing, Commun. ACM 19 (7) (July 1976).
[54] Y.S. Ma, A.J. Offutt, Y.R. Kwon, MuJava: a mutation system for Java, in: ICSE 2006, ACM, 2006.
[55] S. Nica, F. Wotawa, Using constraints for equivalent mutant detection, in: Workshop Formal Methods in the Development of Software, 2012.
[56] A.J. Offutt, W.M. Craft, Using compiler optimization techniques to detect equivalent mutants, Softw. Test. Verif. Reliab. 4 (3) (1994).
[57] A.J. Offutt, Investigations of the software testing coupling effect, ACM Trans. Softw. Eng. Methodol. 1 (1) (1992) 5–20.
[58] A.J. Offutt, S.D. Lee, An empirical evaluation of weak mutation, IEEE Trans. Softw. Eng. 20 (5) (1994) 337–344.
[59] A.J. Offut, G. Rothermel, C. Zapf, An experimental evaluation of selective mutation, in: ICSE 1993, IEEE, 1993.
[60] A.J. Offutt, R.H. Untch, Mutation 2000: uniting the orthogonal, in: Mutation Testing for the New Century, Kluwer Academic Publisher, 2001.
[61] A.J. Offutt, J. Voas, Subsumption of condition coverage techniques by mutation testing, Tech. Report ISSE-TR-96-01, Dpt. Information and Software System Engineering, George Mason Univ., 1996.
[62] G.C. Necula, S. Mcpeak, S.P. Rahul, W. Weimer, CIL: intermediate language and tools for analysis and transformation of C programs, in: CC 2002, Springer, 2002.
[63] M. Papadakis, N. Malevris, An effective path selection strategy for mutation testing, in: APSEC 2009, IEEE, 2009.
[64] M. Papadakis, N. Malevris, Automatic mutation test case generation via dynamic symbolic execution, in: ISSRE 2010, IEEE, 2010.
[65] M. Papadakis, N. Malevris, Automatically performing weak mutation with the aid of symbolic execution, concolic testing and search-based testing, Softw. Qual. J. 19 (4) (2011).
[66] M. Papadakis, N. Malevris, M. Kallia, Towards automating the generation of mutation tests, in: AST 2010 (with ICSE 2010), 2010.
[67] R. Pandita, T. Xie, N. Tillmann, J. de Halleux, Guided test generation for coverage criteria, in: ICSM 2010, IEEE, 2010.
[68] K. Sen, D. Marinov, G. Agha, CUTE: a concolic unit testing engine for C, in: ESEC/FSE 2005, ACM, 2005.
[69] T. Su, G. Pu, B. Fang, J. He, J. Yan, S. Jiang, J. Zhao, Automated coverage-driven test data generation using dynamic symbolic execution, in: SERE 2014, IEEE, 2014.
[70] N. Tillmann, J. de Halleux, Pex-white box test generation for NET, in: TAP 2008, Springer, 2008.
[71] R.H. Untch, A.J. Offutt, M.J. Harrold, Mutation analysis using mutant schemata, in: ISSTA, ACM, 1993.
[72] W.E. Wong, A.P. Mathur, Reducing the cost of mutation testing: an empirical study, J. Syst. Softw. 31 (3) (1995) 185–196.
[73] N. Williams, B. Marre, P. Mouy, On-the-fly generation of K-path tests for C functions, in: ASE 2004, IEEE, 2004.
[74] T. Xie, N. Tillmann, P. de Halleux, W. Schulte, Fitness-guided path exploration in dynamic symbolic execution, in: DSN 2009, IEEE, 2009.
[75] H. Zhu, P.A.V. Hall, J.H.R. May, Software unit test coverage and adequacy, ACM Comput. Surv. 29 (4) (1997).
[76] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. de Halleux, H. Mei, Test generation via dynamic symbolic execution for mutation testing, in: ICSM 2010, IEEE, 2010.
[77] M. Staats, Towards a framework for generating tests to satisfy complex code coverage in Java pathfinder, in: NASA Formal Methods Symposium 2009, Springer, 2009.
[78] M. Staats, C. Pasareanu, Parallel symbolic execution for structural test generation, in: ISSTA 2010, ACM, 2010.