Efficient Leveraging of Symbolic Execution to Advanced Coverage Criteria ⋆

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

***Abstract*—Automatic test data generation (ATG) is a major topic in software engineering. In this paper, we bridge the gap between the coverage criteria supported by state-of-the-art white-box ATG technologies, especially Dynamic Symbolic Execution, and advanced coverage criteria found in the literature. We define a new testing criterion, label coverage, and prove it to be both expressive and amenable to efficient automation. We propose several innovative techniques resulting in an effective black-box support for label coverage, while a direct approach induces an exponential blow-up of the search space. Experiments show that our optimisations yield very significant savings, allowing to leverage ATG to label coverage with only a slight overhead.**

Keywords*—*Testing, symbolic execution, coverage criteria

I. Introduction

Context and problem. Automatic test data generation (ATG) is a major concern in software engineering and program analysis. Recent progress in automated theorem proving led to significant improvements of symbolic approaches for white-box ATG, such as Dynamic Symbolic Execution (DSE) [13], [36], [40]. Tools have been developed [5], [7], [8], [14], [37] and impressive case-studies have been carried out [7], [8], [16].

We consider the case where ATG aims at generating a test suite which is then passed to one or several external oracles, in order to assess for example functional correctness, security or performance. The more behaviours the test suite exercise, the better. A standard way of measuring this diversity passes through coverage criteria [2], [42]. Many such criteria have been defined along the years, from control-flow or data-flow criteria to mutation [11], input domain partitions and MCDC.

DSE mostly follows an exhaustive exploration of the path space of the program under test, covering all execution paths up to a given bound. While this path-oriented criterion proves successful in some contexts, it is well known that the resulting test suite can miss interesting behaviours related to data rather than control. Moreover, standard DSE does not support coverage objectives defined over artifacts not explicitly present in the source code, such as multiple-condition coverage, while they could efficiently guide test generation.

Goal. Our main objective is to bridge the gap between coverage criteria supported by symbolic ATG tools, especially DSE, and advanced coverage criteria found in the literature. Recent works aim at leveraging DSE to other coverage criteria [18], [32], [33], [34], [35], [43] or improving DSE bug-detection abilities by making explicit run-time error conditions [9], [15], [20]. These approaches are mainly based on instrumentation and allow for black-box reuse of existing technologies. However, they come at a high price since they may induce a blow-up of the path space and a significant overhead (a recent paper [18, Table 2] reports on a 272x average time-overhead, with a worst case of 2,000x). We follow the same general line, emphasising black-box reuse as much as possible. However, we focus on two main points mostly left unaddressed: we want to formally characterize the class of coverage criteria that can be supported by DSE-like techniques, and we want to support it *efficiently*.

Approach. We define *label coverage*, a new testing criterion which appears to be both expressive and amenable to efficient automation. Especially, it turns out that *DSE can be extended for label coverage with only a slight overhead*. Labels are predicates attached to program instructions through a labelling function. A label is covered if a test execution reaches it and satisfies the predicate. This idea underlies former work [9], [15], [18], [20], [34], [43]. We generalize these results and propose ways of taming the potential blow-up. Especially, we introduce *tight instrumentation*, where "tight" is made precise in the paper, and a strong coupling of DSE and label coverage named *iterative label deletion*. Their combination results in an effective support for label coverage in DSE. Besides, both techniques can be implemented in black-box.

Contribution. Our main contributions are the following:

- We show that label coverage is expressive enough to faithfully emulate many standard coverage criteria, from decision or condition coverage (Theorem 1) to a substantial subset of weak mutations (the side-effect free fragment, Theorem 2). Labels can be seen in some way as a convenient and powerful specification mechanism for coverage criteria.
- We formally characterise the properties of direct instrumentation for label coverage. The instrumentation is sound w.r.t. label coverage and leads to very efficient coverage score computation. However, it yields an exponential increase as well as a "complexification" of the path space (Theorem 5).
- We propose *tight instrumentation* and *iterative label deletion* as ways of taming this complexity blow-up. Tight instrumentation yields only a linear growth of the path space without any complexification (Theorem 7). Both techniques are orthogonal and allow for

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

a significant speed-up. Moreover, they can both be implemented either through dedicated DSE algorithms or in a black-box manner.

- We have implemented these results inside a DSE tool [40]. Initial experiments show that our optimisations yield very significant reductions of both search space and computation time compared to direct instrumentation (several-orders-of-magnitude speedup in some cases). It follows that ATG for label coverage can be achieved at a very reasonable cost w.r.t. usual DSE.

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 as well as cheap coverage score computation. Moreover, static analysis techniques can also be used directly on the instrumented programs in order to detect uncoverable labels, as was proposed for mutation testing [24].

This work 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 more advanced testing criteria, such as side-effect free weak mutations. On the other hand, we identify a large subclass of weak mutations amenable to efficient automation, both in terms of ATG and mutation score computation.

Outline. After presenting basic notation (Section II), we define labels and explore their expressiveness (Section III). We then focus on automation. The direct instrumentation is defined and studied (Section IV). Afterwards, we describe our own approach to label-based ATG (Section V) and present first experiments (Section VI). Finally, we sketch a highly automatized testing framework based on labels (Section VII), discuss related work (Section VIII) and give a conclusion (Section IX).

II. Background

A. Notation

Given a program P over a vector V of m input variables taking values in a domain $D \triangleq D_1 \times \ldots \times D_m$, 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 $t \rightsquigarrow_P (loc, S)$, 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.

Given a test objective $\mathbf{c}$, we write $t \rightsquigarrow_P \mathbf{c}$ 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 $TS \rightsquigarrow_P \mathbf{C}$ when for any $\mathbf{c} \in \mathbf{C}$, there exists $t \in TS$ such that $t \rightsquigarrow_P \mathbf{c}$. These definitions are generic and leave the exact definition of "covering" to the considered testing 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 $t \rightsquigarrow_P \mathbf{c}$ if t reaches some (loc, S) where `cond` evaluates to *true* (resp. *false*) in S. Coverage criteria used through the paper and their associated notions of covering are described in Section III-B.

B. DSE in brief

We recall here a few basic facts about Symbolic Execution (SE) [21] and Dynamic Symbolic Execution (DSE) [13], [36], [40]. Let us consider a program under test P with input variables V over domain D and a path σ of P. The key insight of SE is that it is possible in many cases to compute a *path predicate* ϕ_σ for σ such that for any input valuation $t \in D$, we have: t satisfies ϕ_σ iff $P(t)$ covers σ. In practice, path predicates are often under-approximated and only the left-to-right implication holds, which is already fine for testing: SE outputs a set of pairs (t_i, σ_i) such that each t_i is ensured to cover the corresponding σ_i. Hence, SE is *sound* from a testing point of view. DSE enhances SE by interleaving concrete and symbolic executions. The dynamically collected information can help the symbolic step, for example by suggesting relevant approximations.

A simplified view of SE is depicted in Algorithm 1. While high-level, it is sufficient to understand the rest of the paper. We assume that the set of paths of P, denoted $Paths(P)$, is finite. In practice, DSE tools enforce this assumption through a bound on path lengths. We assume the availability of a procedure for path predicate computation (with predicates in some theory T), as well as the availability of a solver taking a formula $\phi \in T$ and returning either *sat* with a solution t or *unsat*. All DSE tools rely on such procedures. The algorithm builds iteratively a test suite TS by exploring all paths.

Algorithm 1: Symbolic Execution algorithm

```
Input: a program P with finite set of paths Paths(P)
Output: TS, a set of pairs (t, σ) such that P(t) ⇝_P σ
1  TS := ∅;
2  S_paths := Paths(P);
3  while S_paths ≠ ∅ do
4      choose σ ∈ S_paths; S_paths := S_paths \ {σ} ;
5      compute path predicate φ_σ for σ ;
6      switch solve(φ_σ) do
7          case sat(t): TS := TS ∪ {(t, σ)};
8          case unsat: skip ;
9      endsw
10 end
11 return TS;
```

The major issue here is that SE and DSE must in some ways explore all $Paths(P)$. Advanced tools explore this set lazily, yet they still have to crawl it. Therefore, the size of $Paths(P)$, denoted $|Paths(P)|$, is one of the two major bottlenecks of SE and DSE, the other one being the average cost of solving path predicates.

Note that Bounded model checking (BMC) [10] is sensitive to the same parameters, as it amounts to building a large formula encompassing all paths up to a given length.

III. Label coverage

A. Definitions

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

- φ contains only variables and expressions defined in P at location loc;
- φ contains no side-effect expressions.

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* $l \triangleq (loc, \varphi)$, denoted $t \rightsquigarrow_{\langle P,L\rangle} l$, if t covers some (loc, S) with S satisfying predicate φ. We say that a test suite satisfies the *label coverage* testing criterion, denoted by **LC**, if it covers all labels in L.

For simplicity, we consider in the rest of the paper *normalized programs*, i.e. programs such that no side-effect occurs in any condition of a branching instruction. This is not a severe 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 [30] frequently used by DSE tools for C programs [36], [40].

B. Expressiveness of label coverage

We seek to characterize the power of the **LC** testing criterion. A key notion is that of *labelling function*. A labelling function ψ 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 labelling function ψ such that for any program P and any test suite TS, we have $TS \rightsquigarrow_P$ **C** iff $TS \rightsquigarrow_{\psi(P)}$ **LC**.

We show first how **LC** can simulate basic graph and logic coverage criteria [2], [42]: instruction coverage **IC**, decision coverage **DC**, condition coverage **CC** (covering all atomic conditions appearing in each decision, and their negations), decision-condition coverage **DCC** (basically, **DC** plus **CC**) and multiple-condition coverage **MCC** (covering all combinations of atomic conditions). The basic idea is to introduce in P labels based on branching predicates and their atomic conditions. An example for **CC** is depicted in Figure 1, where additional labels (right) enforce coverage of the two atomic conditions `x==y` and `a<b`.

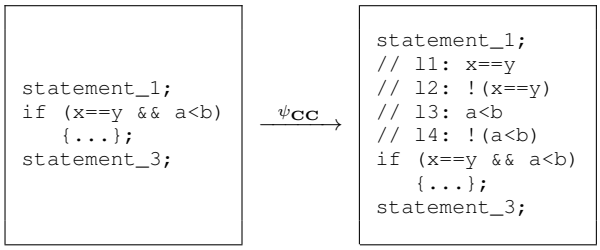

Fig. 1. Simulating **CC** with labels

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

Proof: We need to define a suitable labelling function for any of the considered criteria. For **IC**, we choose the labelling function $\psi_{\mathbf{IC}}$ adding all labels of the form $(loc, true)$, where loc is any location of P. Given a test suite TS, $TS \rightsquigarrow_P$ **IC** iff TS can reach any loc of P iff TS covers any $(loc, true)$ iff $TS \rightsquigarrow_{\psi_{\mathbf{IC}}(P)}$ **LC**. We conclude that **IC** can be simulated by **LC**. Other criteria are handled similarly. The labelling function $\psi_{\mathbf{DC}}$ adds the set of all (loc, φ) and $(loc, \neg\varphi)$, where loc contains a conditional statement with condition φ. $\psi_{\mathbf{CC}}$ 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 the a_i are atomic conditions and $\bar{a}_i$ denotes either a_i or $\neg a_i$. ■

Weak mutations. We now consider a more involved testing criterion, namely weak mutations. In mutation testing [11], test objectives consist of *mutants*, i.e. slight syntactic modifications of the program under test. 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** [17], a mutant M is covered by t, denoted $t \rightsquigarrow_P M$, if the internal states of $P(t)$ and $M(t)$ differ from each other right after the mutated location (cf. Figure 2). **M** is a powerful testing criterion in practice [1], [28]. While less powerful in theory, **WM** appears to be almost equivalent to **M** in practice [26].

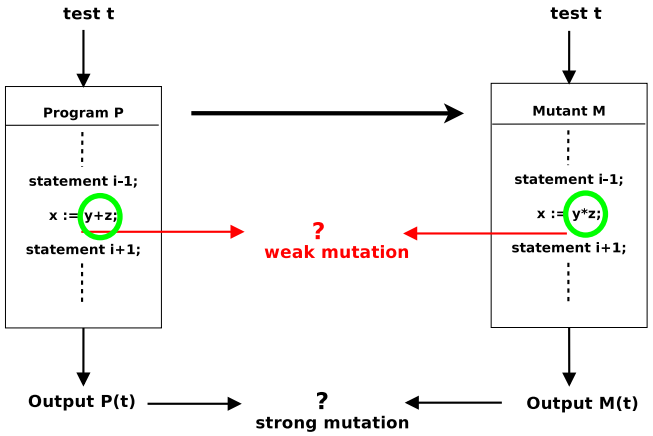

Fig. 2. Strong and weak mutations

We show hereafter that a substantial part of **WM** can be simulated by **LC**. First we need a few more definitions. Mutation testing is parametrised by a set of *mutation operators* O. A 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 (atomic mutation). We denote $\mathbf{WM}_O$ the weak mutation criterion resticted to mutants created through operators in O. We consider that mutations can affect either a left-hand side value (lhs), an expression or a condition. This is a very generic model of mutations, encompassing all standard operators [2]. Finally, we restrict ourselves to mutation operators neither affecting nor introducing side-effect expressions (including calls to side-effect prone functions). We refer to such operators as *side-effect free mutation operators*.

Theorem 2: For any finite set O of side-effect free mutation operators, $\mathbf{WM}_O$ can be simulated by **LC**.

Proof: 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 Figure 3. 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 $t \rightsquigarrow_P M$ iff $t \rightsquigarrow_{\psi(P)} l$. Note that $t \rightsquigarrow_{\psi(P)} l$ 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 to $t \rightsquigarrow_P M$.
- `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, there are only two ways of noticing a difference: either the old value of `lhs` differs from `expr`, then `lhs` will evaluate to different values in $P(t)$ (equals to `expr`) and in $M(t)$ (remains unchanged) just after the mutation, or the symmetric counterpart for `lhs'`. This is exactly what l encodes.

By iterating this technique on all mutants created by the considered mutation operators, we obtain the desired labelling function. ■

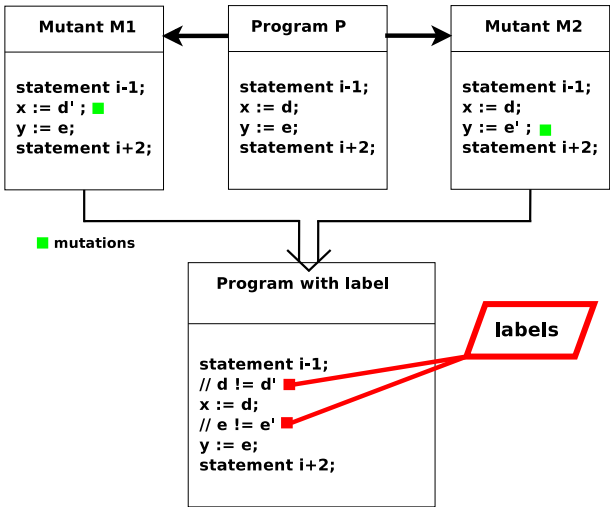

Fig. 3. Simulating weak mutants with labels

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 proved to be almost as powerful as high-order mutations [25]. 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 [2], which have been shown mostly equivalent to much larger sets of operators [27], [39]. It is left as an open question to quantify more precisely what is lost with restriction (2.b). Still, the previous points show that side-effect free **WM** is a substantial (strict) subset of **WM**.

Other criteria. Several other testing criteria commonly found in the literature can be emulated by labels. For example:

- Input Domain Coverage: assuming a partition of the input domain D of P given as disjoint predicates $\varphi_1, \ldots, \varphi_k$, this criterion consists of considering one t_i for each φ_i. The corresponding labelling function adds all labels of the form (loc_0, φ_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 [2].
- Run-Time Error Coverage: test objectives corresponding to *run-time errors* such as those implicitly searched for in active testing or assertion-based testing [9], [15], [20] 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 φ_{safe} will be tagged by a label $(loc, \neg\varphi_{\text{safe}})$.

Limits. The following criteria cannot be emulated through labels, at least with simple encoding: weak mutations with operators involving side-effects, criteria imposing constraints on paths rather than constraints on program locations (k-path coverage, data-flow criteria) and criteria involving different paths (MCDC) or even different programs (strong mutations). It is left as future work to study whether these limitations are strict or not.

IV. Automating LC: A First Attempt

Given an annotated program $\langle P, L \rangle$, we need automatic methods for: (1) computing the **LC** score of a given test suite TS, and (2) deriving a test suite achieving high **LC**-coverage. 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 [9], [15], [18], [20], [34], [35], [43]. While it allows for cheap **LC** score computation, it is far from efficient for ATG, mainly because of an exponential blow-up of the path space of the program.

A. 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 (`φ`) {};` such that all instructions leading to *loc* in P lead to I in P', and I leads to *loc*. The transformation is depicted in Figure 4. When different labels

are attached to the same location, the new instructions are chained together in a sequence ultimately leading to loc.

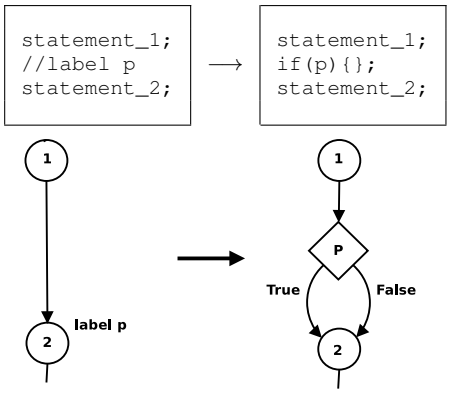

Fig. 4. Direct instrumentation P'

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

Theorem 3 (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 4: 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 $|TS| \cdot maxtime(\{P'(t)|t \in TS\})$.

Computing **LC** score can be done independently from $|L|$. Regarding coverage score computation, **LC** is much closer to **DC** (each test t is executed only once) than it is to **WM** (each test t is executed *once per mutant*). While efficient score computation is a difficult issue in mutation testing, Theorems 2 and 4 show that the side-effect free subset of **WM** does enjoy efficient score computation.

B. Drawbacks

So far, the direct instrumentation seems to perfectly suit our needs. Unfortunately, it is significantly inefficient for ATG. There are two main reasons for that.

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

(‡) DSE will naturally produce a test suite covering several times the same labels, which is useless since each label needs to be covered only once.

We 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 paths in $Paths(P')$. Let us call *label constraints* all additional branches φ and $\neg\varphi$ introduced in P' compared to P, and let us denote by m the maximal number of labels *per* location in P. 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 σ. 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) σ does not pass through any label constraint. Theorem 5 summarises these results.

Theorem 5 (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.

Both aspects are problematic for symbolic exploration of the search space: more paths means either more requests to a theorem prover (DSE) or a larger formula (BMC), while more constrained paths means more expensive requests.

V. Efficient ATG for LC

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*.

A. Tight instrumentation

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

- label constraint φ 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 Figure 5: for each label (loc, φ), we introduce a new instruction `if (nondet) {assert(`φ`); exit};` where `assert(`φ`)` requires φ to be verified, `exit` forces the execution to stop and `nondet` is a non-deterministic choice[1].

In the resulting instrumented program $P^\star$ (Figure 5, right column), when an execution reaches loc, it gives rise to two execution paths: the first one tries to cover the label by asserting φ and *stops right there*, the second one simply follows its execution *as it would do in* P, neither φ nor $\neg\varphi$ being enforced.

Let us denote by **NA** the test objective over $P^\star$ requiring to cover all **N**ew **A**ssert introduced by the instrumentation (with condition evaluating to true). Tight instrumentation is sound w.r.t. **LC** in the following sense.

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

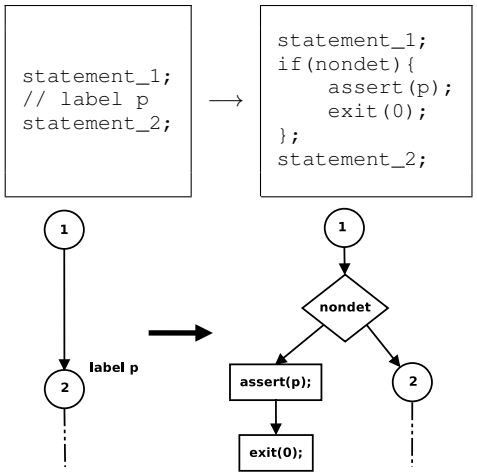

Fig. 5. Tight instrumentation $P^\star$

Theorem 6 (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**.

Interestingly, tight instrumentation does not show any of the issues reported in Theorem 5. The underlying reasons have been sketched at the beginning of Section V-A and are depicted in Figure 6. A single execution path in P going through n labels can give birth up to 2^n paths in P' (left column), while it can create only $n+1$ paths in $P^\star$ (right column). Moreover, each path in $P^\star$ can go through at most one single positive label constraint, while a path σ' in P' can carry up to $|\sigma'|$ (positive or negative) label constraints. These results are summarized in Theorem 7.

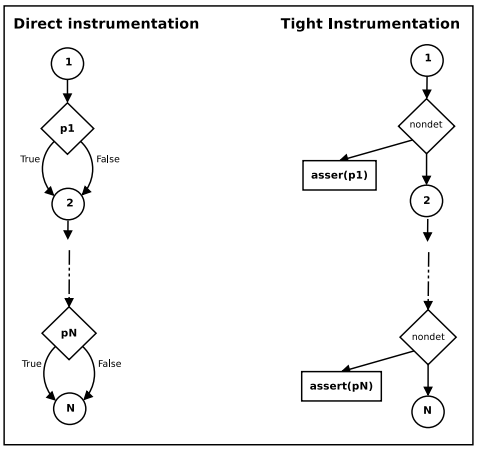

Fig. 6. Direct vs. tight instrumentation

Theorem 7 (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 $m \cdot k$;
- any $\sigma \in Paths(P^\star)$ carries at most one label constraint.

Proof: The main reasons behind this result directly follow from the tight instrumentation, and have already been exposed just before Theorem 7. We can be more precise: $|Paths(P^\star)|$ is bounded by $(m \cdot k + 1) \cdot |Paths(P)|$. ■

Theorems 5 and 7 imply that any path-based program analysis conducted over $P^\star$ will have a much easier task than if conducted over P', since $P^\star$ contains exponentially less paths and those paths are simpler.

B. Iterative label deletion

We focus now on issue (‡) pointed out in Section IV-B. 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 path prefixes ending in an already-covered `assert(`φ`)`. Whether they fail or not, these computations are redundant since Theorem 6 only requires that each new `assert` be covered once.

Iterative label deletion (IDL) consists in (conceptually) erasing a label constraint as soon as it is covered, so that it will not affect the subsequent path search. IDL requires to modify SE/DSE in the following way: each label l is equipped with a boolean variable b_l set to true iff l has already been covered during path exploration, and attempts to symbolically execute paths leading to l continue as long as b_l is false.

We present DSE with IDL over annotated programs in Algorithm 2, denoted DSE$^\star(\langle P, L\rangle)$, where modifications w.r.t. standard SE/DSE are pointed out by $(\star)$ marks. We assume that $Paths(\langle P, L\rangle)$ is constructed in the following way: at each step, a run encountering a label $l \triangleq (loc, \varphi)$ can either choose to go through l (enforcing φ) and continue, or bypass l (no constraint) and continue.

Algorithm 2: Symbolic Execution with IDL

Input: an annotated program $\langle P, L\rangle$ with finite set of paths $Paths(P)$
Output: TS, a set of pairs (t, σ) such that $P(t) \rightsquigarrow_P \sigma$

```
1  TS := ∅;
2  S_paths := Paths(⟨P, L⟩);
3  while S_paths ≠ ∅ do
4      choose σ ∈ S_paths; S_paths := S_paths \ {σ} ;
5      compute φ_σ;
6      switch solve(φ_σ) do
7          case sat(t):
8              TS := TS ∪ {(t, σ)};
9              (⋆) for all l covered by σ, do: b_l := 1 ;
10             (⋆) remove from S_paths all σ' going through
11                 a label l s.t. b_l = 1 ;
12         case unsat: skip ;
13     endsw
14 end
15 return TS;
```

For integration in a realistic SE/DSE setting with dynamic exploration of path space, we distinguish two flavors of IDL:

IDL-1 a label is marked as covered only when it belongs to a path prefix being successfully solved. This is a purely symbolic approach.

IDL-2 a label is also marked when it is covered by a concrete execution, taking advantage of dynamic runs to delete several labels at once.

Combining IDL with tight instrumentation. Both variants of IDL can be combined with tight instrumentation either in a dedicated manner or in a black-box setting. Since dedicated implementations are straightforward, we focus hereafter on black-box implementations. An instrumentation enforcing IDL-1 over $P^\star$ is depicted in Figure 7. We add extra boolean variables for coverage, denoted `b_l` where `l` is a label identifier. Yet, it is mandatory that the coverage information be global to the whole search rather than bound to a single path, and that the `b_l` be treated concretely by DSE. For simplicity, we assume that it is achieved by putting the coverage information in an external file, accessed and modified through (concretized only) operations `read(b_l)` and `set_covered(b_l)`.

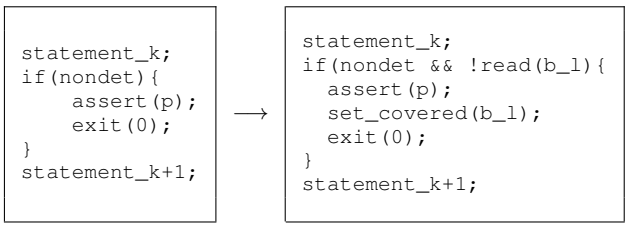

Fig. 7. IDL-1 variant of tight instrumentation $P^\star$

Enforcing IDL-2 in a black-box setting requires a fine-grained control over the DSE procedure. We need to be able to query the DSE engine for the next generated test datum. The procedure reuses the IDL-1-variant of tight instrumentation $P^\star$, but each new generated test datum t is also run on the *direct instrumentation* P'. Unlike $P^\star$, P' does not exit at the first covered label, so all labels covered by t will be marked in the coverage file before the next test datum is searched for. The technique is depicted in Figure 8.

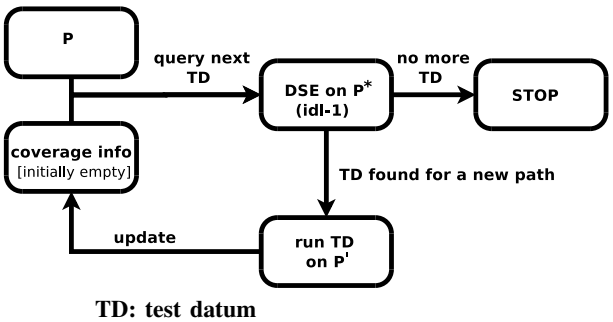

Fig. 8. IDL-2 variant for DSE

We denote by DSE$^\star(P^\star)$ the combination of tight instrumentation and IDL presented in Figure 8, and we denote by DSE$(\langle P, L\rangle)$ the variant of Algorithm 2 where lines 9-11 are removed. Considering only deterministic DSE techniques, the following result holds.

Theorem 8 (Relative completeness): Given an annotated program $\langle P, L\rangle$ and its tight instrumentation $P^\star$, then both DSE$^\star(\langle P, L\rangle)$ and DSE$^\star(P^\star)$ cover as many labels as DSE$(\langle P, L\rangle)$ does.

Proof: We first show that discarding paths going through an already covered label in DSE$^\star(\langle P, L\rangle)$ (cf lines 9-11 of Algorithm 2) does not decrease label coverage w.r.t standard DSE on $\langle P, L\rangle$. Given a label l, then for any path $\sigma \in Paths(\langle P, L\rangle)$ covering l, there exists by definition of $Paths(\langle P, L\rangle)$ another path σ' passing through l and the same statements in P as σ while bypassing any other label, so that σ covers l iff σ' covers l. Since σ' cannot be discarded in DSE$^\star(\langle P, L\rangle)$ unless l is covered, it follows that the coverage cannot decrease, proving that DSE$^\star(\langle P, L\rangle)$ and DSE$(\langle P, L\rangle)$ achieve the same coverage. Regarding coverage equivalence between DSE$^\star(P^\star)$ and DSE$(\langle P, L\rangle)$, we reduce it to proving coverage equivalence between DSE$^\star(P^\star)$ and DSE$(P^\star)$ through Theorem 6. We can then conclude with a similar proof technique. ■

VI. Implementation & Experiments

A. Implementation

We have implemented tight instrumentation and iterative label deletion inside PathCrawler [40]. The tool targets safety-critical C programs, with a strong focus on relative completeness guarantees. For example, the underlying constraint solver deals precisely with modular arithmetic, bitwise operations, floats and multi-level pointer dereferences. The DSE engine relies on a basic depth-first search and is highly optimised for programs with many infeasible paths [3].

Our implementation follows the description of Section V. We implement tight instrumentation and IDL-2 in a dedicated manner rather than a full black-box approach because PathCrawler does not offer yet the required API for IDL-2. The search heuristics is mostly depth-first, but labels are handled as soon as possible.

B. Experiments

Preliminary experiments have been conducted in order to investigate the following properties: (i) the relative gain of our two optimisations w.r.t. direct instrumentation, (ii) the overhead of leveraging DSE to **LC**. Evaluating the practical feasibility of label-based DSE over very large programs or its bug-finding power are left as future work[2].

Protocol. We consider standard benchmark programs[3] taken from related works [9], [34], [32], 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 III-B). For **WM**, we mimic mutations introduced by MuJava [22] for operators AOIU, AOR, COR and ROR [2][4]. Annotation is done manually[5]. Uncoverable labels (typically coming from equivalent mutants) are not discarded.

We compare the following algorithms: DSE(P) denotes standard DSE (witness), DSE(P') denotes standard DSE on direct instrumentation, DSE$(P^\star)$ denotes standard DSE on tight instrumentation and DSE$^\star(P^\star)$ denotes DSE with iterative label deletion run on tight instrumentation. The DSE engine runs in deterministic mode, generating the same concrete

[2]Bug-finding power of criteria in Section III is already extensively studied.
[3]Available at `http://sebastien.bardin.free.fr/benchs-icst.zip`
[4]These operators are considered very powerful in practice [27], [39].
[5]The cost of automatic annotation is negligible w.r.t. the cost of DSE.

values from one run to the other. Time-out for solver is set to 1 min, time-out for test generation is set to 1h30. Experiments are performed on an Intel Core2 Duo 2.40GHz, 4GB of RAM.

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 leveraging DSE to labels and the benefits of our optimisations. Coverage score together with computation time indicate how practical label-based DSE is.

It must be highlighted that PATHCRAWLER does not stop until all feasible paths are explored. This strategy gives us a good estimation of the size of the path space, however in pratice it would be wiser to implement a label-based stopping criterion. Hence, from a feasibility point of view, results reported here are too pessimistic.

Results. A representative subset of results is presented in Table I. 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 17/25 examples). We also observe that direct instrumentation yields a significant overhead, confirming previous work [18]: DSE(P') has four time-outs (TO) while DSE(P) has none, time-overhead goes up to 122x (excluding TO), growth of the path-space reaches 50x.

On the other hand, tight instrumentation DSE*($P^\star$) yields only a very reasonable overhead w.r.t. standard DSE: no time-out is reported, time-overhead is kept under 7x with an average of 2.4x, growth of the path-space is limited to 3x. On some examples, tight instrumentation performs remarkably better than direct instrumentation (94s vs TO on `gd_5-wm`). Interestingly, DSE*($P^\star$) does perform better than standard DSE (up to 2x) on a few examples with very few additional paths. We conjecture that additional label constraints may sometimes greatly simplify the solving process, but it must be investigated further.

Finally, as expected, DSE($P^\star$) stands between DSE(P') and DSE*($P^\star$) for the number of paths. Results are more mitigated for computation time, where DSE($P^\star$) is slower than DSE(P') on several examples, probably due to the fact that our implementations of DSE($P^\star$) and DSE*($P^\star$) are not optimal.

Conclusion. These experiments confirm our formal predictions:

- fully-optimised DSE performs significantly better on difficult programs than the direct instrumentation, both in terms of search space and computation time;
- the overhead w.r.t. standard DSE turns out to be always acceptable, and often very low.

These results suggest that DSE can be efficiently leveraged to **LC** coverage thanks to our optimisations. Further experiments on larger programs are required to fully confirm that point.

VII. BEYOND TEST DATA GENERATION

Section III proves that **LC** is a powerful coverage criterion, encompassing many standard criteria and a large subset of weak mutations. Section V and Section VI demonstrate the feasibility of efficient ATG for **LC**, with a cost-effective integration in DSE. We also sketched in Section IV how to perform

			DSE(P) (witness)	DSE(P')	DSE($P^\star$)	DSE*($P^\star$)
trityp	cc	#paths	35	183	83	46
50 loc	24 l	time	1.3s	1.6s	2s	4.5s
		cover		24/24	24/24	24/24
	mcc	#paths	35	337	110	66
	28 l	time	1.3s	1.9s	3s	2.1s
		cover		28/28	28/28	28/28
	wm	#paths	35	x	506	48
	129 l	time	1.3s	x	12s	5.1s
		cover		x	120/129	120/129
4balls	wm	#paths	7	**195**	75	**23**
35 loc	67 l	time	1.2s	1.9s	2.1s	2.1s
		cover		56/67	56/67	56/67
utf8-3	wm	#paths	134	1,379	626	313
108 loc	84 l	time	1.4s	4.2s	4.3s	3.8s
		cover		55/84	55/84	55/84
utf8-5	wm	#paths	680	**11,111**	3,239	**743**
108 loc	84 l	time	2s	40s	24s	8.1s
		cover		82/84	82/84	82/84
utf8-7	wm	#paths	3,069	**81,133**	14,676	**3,265**
108 loc	84 l	time	5.8s	**576s**	110s	**35s**
		cover		82/84	82/84	82/84
tcas	cc	#paths	2,787	3,508	3,508	2,815
124 loc	10 l	time	2.9s	3.6s	5s	3.4s
		cover		10/10	10/10	10/10
	mcc	#paths	2,787	3,988	3,988	3,059
	12 l	time	2.9s	4.2s	5.2s	3.9s
		cover		11/12	11/12	11/12
tcas'	wm	#paths	4,420	**300,213**	20,312	**6,014**
124 loc	111 l	time	5.6s	**662s**	120s	**27s**
		cover		101/111	101/111	101/111
replace	wm	#paths	866	**87,498**	6,420	**2,347**
100 loc	79 l	time	2s	**245s**	64s	**14s**
		cover		70/79	70/79	70/79
full_bad	cc	#paths	2,563	5,148	5,129	3,209
219 loc	16 l	time	5s	8s	14s	7s
		cover		12/16	12/16	12/16
	mcc	#paths	2,563	12,360	12,296	7,043
	39 l	time	5s	19s	32s	19s
		cover		24/39	24/39	24/39
	wm	#paths	2,593	19,336	10,610	5,414
	46 l	time	5s	35s	40s	19s
		cover		34/46	34/46	34/46
get_tag-5	cc	#paths	11,833	40,102	22,669	11,843
240 loc	20 l	time	60s	210s	651s	64s
		cover		20/20	20/20	20/20
	mcc	#paths	11,833	41,605	23,794	11,848
	26 l	time	60s	100s	510s	48s
		cover		26/26	26/26	26/26
	wm	#paths	11,833	58,646	28,919	11,856
	47 l	time	61s	140s	719s	51s
		cover		44/47	44/47	44/47
get_tag-6	cc	#paths	76,456			76,468
240 loc	20 l	time	3,011s	**TO**	TO	**1,512s**
		cover				20/20
	wm	#paths	76,456			76,481
	47 l	time	3,011s	**TO**	TO	**1,463s**
		cover				44/47
gd-5	cc	#paths	14,516	18,220	17,018	14,605
319 loc	36 l	time	52s	66s	91s	59s
		cover		36/36	36/36	36/36
	mcc	#paths	14,516	20,261	18,799	15,201
	36 l	time	51s	71s	101s	80s
		cover		29/36	29/36	29/36
	wm	#paths	14,516			14,607
	63 l	time	50s	**TO**	TO	**94s**
		cover				62/63
gd-6	cc	#paths	107,410	131,726	125,024	107,500
319 loc	36 l	time	3,740s	3,816s	5,534s	2,945s
		cover		36/36	36/36	36/36
	mcc	#paths	107,410	144,840	137,328	111,208
	36 l	time	3,740s	3,822s	6,281s	3,447s
		cover		29/36	29/36	29/36
	wm	#paths	107,410			107,521
	63 l	time	3,740s	**TO**	TO	**2,232s**
		cover				63/63

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

TABLE I. EXPERIMENTAL RESULTS FOR ATG

cheap **LC** score computation. Everything put together, labels form the basis of a very powerful framework for automatic testing, handling many different criteria in a uniform fashion. We describe such a view in Figure 9. Starting from a program P and a testing criterion $\mathbf{C}$, a predefined labelling function $\psi_{\mathbf{C}}$ creates the $\mathbf{C}$-equivalent annotated program $\langle P, L\rangle$ (Theorems 1 and 2). Then, we can perform efficient **LC** score computation and **LC**-based ATG through instrumentation (Theorems 4 and 6). Finally, static analysis techniques can be used on $P^\star$ in order to detect uncoverable labels, i.e. labels $l \triangleq (loc, \varphi)$ for which there is no test datum t such that $t \rightsquigarrow_{\psi_{\mathbf{C}}(P)} l$. Static detection of uncoverable labels can help ATG tools by avoiding wasting time on infeasible objectives, as was observed in the case of mutation testing [19].

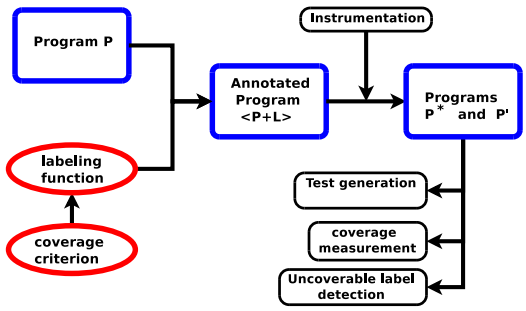

Fig. 9. **LC**-coverage framework

VIII. Related work

Leveraging DSE to higher coverage criteria. The need for enhancing DSE with better coverage criteria has already been pointed out in active testing (a.k.a assertion-based testing) [9], [15], [20], Mutation DSE [32], [33] and Augmented DSE [18], [35], [43]. 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.

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 III. 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 optimisation. In that case only the IDL-1 flavour makes sense since an execution cannot cover two different run-time errors. 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 [9]. They can be reused for labels, and should be useful when many labels are uncoverable.

Following Offut *et al.* [12], Papadakis *et al.* show that **WM** can be reduced to branch coverage through the use of a variant of Mutant Schemata [38]. 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 [32] based on a direct encoding similar to our DSE(P') scheme, and a more ad hoc approach [32] preventing reuse of existing DSE tools but offering several optimisations. Papadakis *et al.* propose a variant of IDL, a dedicated search heuristic based on shortest paths [31] 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 [18] is a variant of direct instrumentation. Several coverage criteria are encoded, getting results similar to those of Section III-B, yet the side-effect free subset of **WM** is not identified. Experiments [18, Table 2] report an average time-overhead of 272x, going up to >2,000x. That confirms the strong benefits of our optimisations, that yield a maximal overhead of 7x.

We give a more generic view of the problem, identifying labels and annotated programs as the key concept underlying the approach. We also clearly identify the limits and hypotheses of the method by defining 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 important, we propose the tight instrumentation which completely prevents complexification. Finally, our optimisations can be implemented in a pure black-box setting and we do not impose anything on the search heuristics, keeping room for future improvements.

Labels and optimized DSE. The label-specific optimisations described here can be freely mixed with other DSE optimisations. It is left as future work to explore which optimisations turn out to be the most effective for labels. As already stated, combining static discovery of uncoverable labels with DSE [9] could be useful for often-uncoverable labels, such as those generated for Run-Time Error Coverage or **MCC**. Another promising direction is to adapt DSE search heuristics [41] by taking advantage of the dissimilarities between labels and branches, possibly getting inspiration from [31].

The IDL optimisation shows some similarities with Look-Ahead pruning (LA) [4], [6]. Basically, LA takes advantage of (global) static analysis to prune path prefixes which cannot reach any uncovered branches. On $P^\star$, IDL-1 is a very specific (but cheap) case of LA while IDL-2 is orthogonal: LA prunes those "label paths" pruned by IDL-1 plus other paths leading only to already covered labels, while IDL-2 prunes several "label paths" at once thanks to dynamic analysis.

Automation of mutation testing. Mutation coverage [11], [28] has been established as a powerful criterion through several experimental studies [1], [28]. Yet, it is very difficult to automatize. Even mutation score computation is expensive in practice if not done wisely. Weak mutations [17] 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 [26], and from a theoretical point of view **WM** subsumes many other criteria [29].

The few existing symbolic methods for mutation-based ATG are based on the encoding proposed by Offutt *et al.* and have already been discussed [12], [34], [33]. The Mutation Schemata technique [38] 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 [24], [23] in a way similar to what is sketched in Section VII.

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 [27], [39] or smart use of operators [19].

IX. Conclusion

Label coverage is a new testing criterion which appears to be both expressive and amenable to efficient automation. Some of the ideas behind labels underly previous work by other teams. We generalise 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 only a slight overhead. Experiments show that our optimisations yield very significant improvements.

This work 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 testing criteria, including side-effect free weak mutations. On the other hand, we identify a powerful criterion amenable to efficient automation, both in terms of ATG and coverage score computation.

References

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

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

[3] 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

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

[5] S. Bardin, P. Herrmann. OSMOSE: Automatic Structural Testing of Executables. Softw. Test., Verif. Reliab. 21(1): 29-54(2011)

[6] J. Burnim, K. Sen. Heuristics for Scalable Dynamic Test Generation. In: ASE 2008. IEEE

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

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

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

[10] E. M. Clarke, D. Kroening, F. Lerda: A Tool for Checking ANSI-C Programs. In: TACAS 2004. Springer

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

[12] R. A. DeMillo, A. J. Offutt: Constraint-Based Automatic Test Data Generation. IEEE Trans. Software Eng. 17(9), 1991

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

[14] P. Godefroid, M. Y. Levin and D. Molnar. Automated Whitebox Fuzz Testing. In: NDSS 2008.

[15] P. Godefroid, M. Y. Levin, D. Molnar: Active property checking. In: EMSOFT 2008. ACM

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

[17] W. E. Howden: Weak mutation testing and completeness of test sets. In: IEEE Transactions on Software Engineering, 8(4). 1982

[18] K. Jamrozik, G. Fraser, N. Tillmann and J. de Halleux. Generating Test Suites with Augmented Dynamic Symbolic Execution. In: TAP 2013.

[19] R. Just, G. M. Kapfhammer and F. Schweiggert. Do Redundant Mutants Affect the Effectiveness and Efficiency of Mutation Analysis? In: ICST 2012. IEEE

[20] B. Korel, A. M. Al-Yami: Assertion-Oriented Automated Test Data Generation. In: ICSE 1996. IEEE

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

[22] Y. S. Ma, A. J. Offutt, Y. R. Kwon: MuJava: a mutation system for java. In: ICSE 2006. ACM

[23] S. Nica, F. Wotawa: Using Constraints for Equivalent Mutant Detection. In: workshop Formal Methods in the Development of Software (2012)

[24] A. J. Offutt, W. M. Craft: Using Compiler Optimization Techniques to Detect Equivalent Mutants. Softw. Test., Verif. Reliab. 4(3). 1994

[25] A. J. Offutt: Investigations of the Software Testing Coupling Effect. ACM Trans. Softw. Eng. Methodol. 1(1): 5-20 (1992)

[26] A. J. Offutt, S. D. Lee: An Empirical Evaluation of Weak Mutation. IEEE Trans. Software Eng. 20(5): 337-344 (1994)

[27] A. J. Offut, G. Rothermel, C. Zapf: An experimental evaluation of selective mutation. In: ICSE 1993. IEEE

[28] A. J. Offutt, R. H. Untch: Mutation 2000: uniting the orthogonal. In: Mutation testing for the new century. Kluwer Academic Publisher, 2001

[29] 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

[30] G. C. Necula and S. Mcpeak and S. P. Rahul and W. Weimer CIL: Intermediate language and tools for analysis and transformation of C programs. In: CC 2002. Springer

[31] M. Papadakis, N. Malevris: An Effective Path Selection Strategy for Mutation Testing. In: APSEC 2009. IEEE

[32] M. Papadakis, N. Malevris: Automatic Mutation Test Case Generation via Dynamic Symbolic Execution. In: ISSRE 2010. IEEE

[33] M. Papadakis, N. Malevris: Automatically performing weak mutation with the aid of symbolic execution, concolic testing and search-based testing. Software Quality Journal 19(4), 2011

[34] M. Papadakis, N. Malevris, M. Kallia: Towards Automating the Generation of Mutation Tests. In: AST 2010 (with ICSE 2010).

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

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

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

[38] R. H. Untch, A. J. Offutt, M. J. Harrold: Mutation Analysis Using Mutant Schemata. In: ISSTA. ACM (1993)

[39] W. E. Wong, A. P. Mathur: Reducing the cost of mutation testing: An emprirical study. In: Journal of Systems and Software, 31(3), 185-196.

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

[41] T. Xie, N. Tillmann, P. de Halleux and W. Schulte. Fitness-Guided Path Exploration in Dynamic Symbolic Execution. In: DSN 2009. IEEE (2009)

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

[43] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. de Halleux and H. Mei. Test generation via Dynamic Symbolic Execution for mutation testing. In: ICSM 2010. IEEE