An All-in-One Toolkit for Automated White-Box Testing*

Sébastien Bardin, Omar Chebaro, Mickaël Delahaye, and Nikolai Kosmatov

CEA, LIST, Gif-sur-Yvette, F-91191, France
first.name@cea.fr

Abstract. Automated white-box testing is a major issue in software engineering. Over the years, several tools have been proposed for supporting distinct parts of the testing process. Yet, these tools are mostly separated and most of them support only a fixed and restricted subset of testing criteria. We describe in this paper FRAMA-C/LTEST, a generic and integrated toolkit for automated white-box testing of C programs. LTEST provides a unified support of many different testing criteria as well as an easy integration of new criteria. Moreover, it is designed around three basic services (test coverage estimation, automatic test generation, detection of uncoverable objectives) covering most major aspects of white-box testing and taking benefit from a combination of static and dynamic analyses. Services can cooperate through a shared coverage database. Preliminary experiments demonstrate the possibilities and advantages of such cooperations.

1 Introduction

Automated white-box testing is a major issue in software engineering. Along the years, several tools have been proposed for supporting distinct parts of the testing process, such as test replay, coverage estimation or automatic test generation. Yet, these tools are mostly separated, and most of them support only a fixed and restricted subset of existing testing criteria.

Our main goals are (1) to provide tool support for most steps of the white-box testing process, and (2) to support a large range of coverage criteria and to offer flexible ways of adding new ones. We propose FRAMA-C/LTEST, a *generic* and *integrated* toolkit for automated white-box testing of C programs. It is generic in the sense that it supports a broad class of coverage criteria in a unified way, and integrated in the sense that it covers most major aspects of white-box testing. FRAMA-C/LTEST is implemented on top of the FRAMA-C verification platform [4] and relies on a combination of test generation and static analysis. More precisely:

- LTEST provides three basic services for test automation: coverage estimation, automatic test generation (ATG) and detection of uncoverable test objectives. Moreover, several coverage criteria are already supported, and adding new ones is straightforward. We achieved this by building the tool upon label coverage [2], a specification mechanism allowing to manage many existing criteria in a unified way.

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

M. Seidl and N. Tillmann (Eds.): TAP 2014, LNCS 8570, pp. 53–60, 2014.

- The toolkit is designed around four basic modules (program annotation, coverage estimation, ATG and detection of uncoverable labels) that advantageously combine static and dynamic analysis techniques and communicate through a shared database of coverage information. This modular architecture allows for flexible interactions between modules and gives opportunities for dedicated optimisations.
- We provide a summary of preliminary results demonstrating the benefits of our hybrid analysis approach, typical use-case scenarios and gains of our optimisations.

The paper is organized as follows. Section 2 provides necessary background on labels. An overview of the LTEST platform is given in Section 3, including a description of the provided services, a typical use-case and implementation details. Section 4 presents a summary of experiments. Finally, related work is discussed in Section 5 and Section 6 concludes the paper.

2 Labels

Label coverage [2] provides a convenient and powerful specification mechanism for coverage criteria. *Labels* are predicates attached to program instructions. A program with labels is called an *annotated program*. A label is covered if a test execution reaches it and satisfies the predicate. Labels can faithfully emulate many standard coverage criteria, from decision or condition coverage to a substantial subset of weak mutations, allowing us to manage all of them in a unified way. Basically, for each test objective a new label is added to the program under test, such that covering the label in the annotated program is equivalent to covering the test objective in the program under test. The automatic insertion of adequate labels for a given coverage criterion is performed by a so-called *labelling function*. Several examples are presented in Fig. 1.

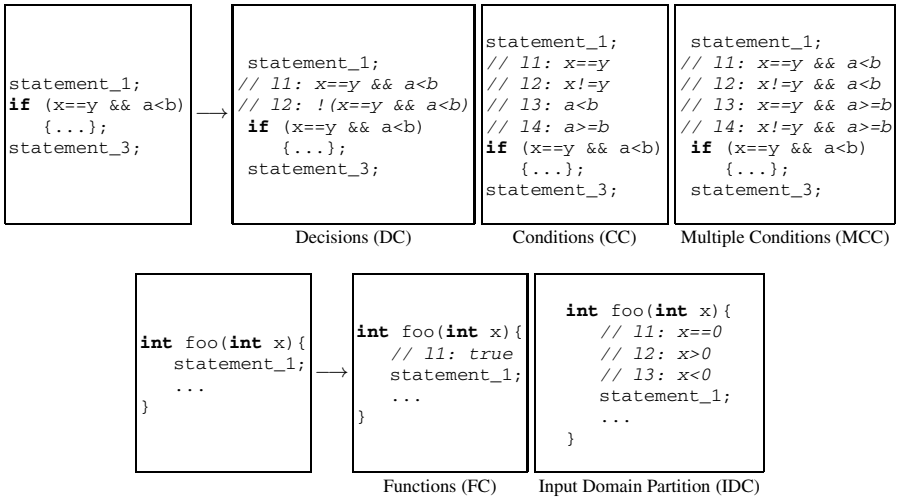

Fig. 1. Simulating standard coverage criteria with labels

Dynamic Symbolic Execution (DSE) [7,11] is a popular approach to automatic test generation, based on path exploration. We showed in previous work [2] how to extend DSE for handling labels with only very small overhead, while prior instrumentation-based approaches incur a blow-up of the search space [9]. We denote by DSE* this modified version of DSE.

3 Overview of the Platform

3.1 From the User Perspective

LTEST comes as a series of FRAMA-C plugins [4]. The toolkit offers the following main services:

Uncoverability Detection: the service detects uncoverable test objectives, i.e. those objectives which cannot be covered by any test datum. The information is primarily used by other modules, but it can also be exported for external use.

Coverage Estimation: the service replays a given test suite and reports its coverage. Coverage is given as a whole (all test objectives taken into account) and per criterion. Moreover, uncoverable or uncovered test objectives are reported.

ATG: the service produces a test suite which can be replayed for coverage estimation. In case a test suite has already been replayed, the ATG service will try to complete the achieved coverage rather than to start from scratch.

The platform currently supports the following test criteria [1,2]: decision coverage (DC), function coverage (FC), condition coverage (CC), multiple-condition coverage (MCC), weak mutation (WM, operators AOR, ROR, COR, ABS) and input domain partition (IDC). 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.

3.2 A Typical Use-Case

To illustrate the usage of the platform, let us consider a toy example. The function `quadrant` of Fig. 2 takes as inputs the coordinates of two points $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ on the plane and checks if they belong to the same quadrant.

Suppose we run LTEST first to generate labels for this function and choose the MCC coverage criterion [1]. Here, 16 labels will be added by the tool before each `if` statement, that is, 64 labels in total. For instance, the labels with the following conditions are added just before line 6 of Fig. 2:

$$x_1 \diamond 0 \ \wedge \ x_2 \diamond 0 \ \wedge \ y_1 \bullet 0 \ \wedge \ y_2 \bullet 0, \quad \text{where } \diamond \in \{\leq, >\}, \bullet \in \{\geq, <\}.$$

Next, we run the ATG service based on DSE*. It covers 58 of the 64 labels after exploring 409 (partial) program paths. The remaining 6 labels are indeed uncoverable. For example, the label $\psi = x_1 > 0 \ \wedge \ x_2 > 0 \ \wedge \ y_1 \geq 0 \ \wedge \ y_2 \geq 0$ added before the statement of line 6 is uncoverable since the condition ψ is weaker than the condition

```
1  // Checks if input points (x1,y1) and (x2,y2) lie in the same quadrant
2  // of the plane. Returns the quadrant number if so, otherwise returns 0.
3  int quadrant (int x1, int y1, int x2, int y2){
4    if(x1 >= 0 && x2 >= 0 && y1 >= 0 && y2 >= 0)
5      return 1;                                    // (+,+): quadrant 1
6    if(x1 <= 0 && x2 <= 0 && y1 >= 0 && y2 >= 0)
7      return 2;                                    // (-,+): quadrant 2
8    if(x1 <= 0 && x2 <= 0 && y1 <= 0 && y2 <= 0)
9      return 3;                                    // (-,-): quadrant 3
10   if(x1 >= 0 && x2 >= 0 && y1 <= 0 && y2 <= 0)
11     return 4;                                    // (+,-): quadrant 4
12   return 0;                                      // not in the same quadrant
13 }
```

Fig. 2. Function `quadrant`

of the first `if` statement followed by a `return` (cf lines 4–5 in Fig. 2), so ψ cannot be satisfied at this program point. Similarly, two uncoverable labels are generated for line 8 and three for line 10, each of them being unsatisfiable because of the preceding `if` and `return` statements. Note that, here, using a standard DSE approach with a direct instrumentation instead of DSE* [2] would lead to exploring 3938 program paths.

To avoid wasting time trying to cover uncoverable labels, we can first run the uncoverability detection service based on static analysis, successfully marking the six uncoverable labels. The ATG service will now ignore them, exploring only 284 program paths (instead of 409) while still covering the same 58 labels.

3.3 Inside LTEST

The toolkit is designed around the notions of labels and annotated programs, and structured in four modules: LANNOTATE, LREPLAY, LUNCOV and LGENTEST. Modules can 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: it annotates the program with labels according to the chosen criteria and creates the status database. The other modules provide user-level services. They can update label statuses and in some cases take advantage of them. Compared to the description given in Sec. 2, labels are equipped with a unique identifier used by the database and a "category" tag allowing their classification according to coverage criteria. Finally, besides annotated programs which are our core language and the primary input for DSE*, we use two closely related classes of instrumented programs for uncoverability detection (Fig. 3(b)) and test replay (Fig. 3(c)).

The whole architecture is depicted in Fig. 4. We give hereafter a few clues about the technologies behind each module.

LANNOTATE: The module implements the idea of labelling functions [2] and can be seen as a mapping

$$(\mathsf{program}, \mathsf{set\ of\ criteria}) \mapsto (\mathsf{annotated\ program}, \mathsf{status\ database}).$$

Given a C program and a set of supported criteria (listed in Sec. 3.1), LANNOTATE automatically computes a program annotated with the labels corresponding to the selected criteria. It also initializes the status database that can be used by other LTEST services.

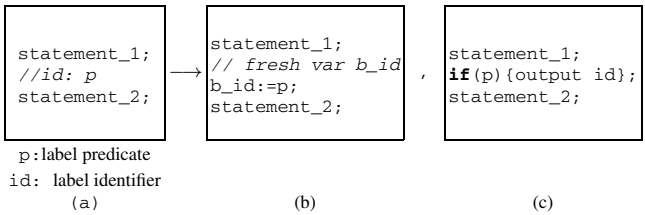

Fig. 3. An annotated program (a), its instrumentations for LUNCOV (b) and LREPLAY (c)

LANNOTATE contains an annotation function per criterion. The module relies on the FRAMA-C kernel services for program transformation [4], based themselves on the CIL library. More precisely, each annotation function takes as input the program's abstract syntax tree (AST), inserts the required labels for the target criterion, and outputs a new AST containing the labels. In addition to already supported criteria, users can extend the module by writing their own annotation functions. LANNOTATE provides facilities to easily insert labels into an AST and to collect all inserted labels into the shared status database. Note that annotated programs can be exported for external use.

LUNCOV: This module acts as a mapping

$$(\text{annotated program}, \text{status database}) \mapsto \text{status database}.$$

Given an annotated program and its status database, LUNCOV runs static analysis to identify uncoverable labels and marks them as uncoverable in the database. A label can be uncoverable for example when it has an unreachable location (dead code) or an unsatisfiable condition.

The implementation of LUNCOV strongly relies on the value analysis plugin (VALUE) of FRAMA-C [4]. VALUE computes (an overapproximation of) the set of possible values of variables at each program point through abstract interpretation. Given an annotated program, we launch VALUE on the instrumented version depicted in Fig. 3(b): if VALUE reports that a "label variable" cannot be true, then the associated label is uncoverable. For example, in Fig. 3(b), if `b_id` cannot be true before `statement_2`, then either the execution cannot reach `statement_2` or the predicate `p` cannot be satisfied. In both cases, it follows that label `id` in the original annotated program of Fig. 3(a) is uncoverable.

LREPLAY: The interface of this module can be seen as a mapping

$$(\text{annotated program}, \text{test suite}, \text{status database}) \mapsto \text{status database}.$$

Given an annotated program and an existing test suite, the module runs each test on the instrumented version of Fig. 3(c) and inspects output traces in order to update label statuses in the status database. In addition, it computes coverage statistics for the given test suite.

LGENTEST: This module provides the test generation service of LTEST and performs the mapping

$$(\text{annotated program}, \text{status database}) \mapsto (\text{test suite}, \text{status database}).$$

LGENTEST implements DSE* [2] and is based on a modified version of the PATHCRAWLER test generator [11]. Compared to [2], the current version includes two new optimisations: (**OPT-1**) DSE* is stopped once all (potentially coverable) labels have been covered, and (**OPT-2**) already-covered labels (e.g. by another test suite) and uncoverable labels are ignored by DSE*. In this way, test generation effectively benefits from static analysis results computed by LUNCOV (cf Sec. 3.2 & 4).

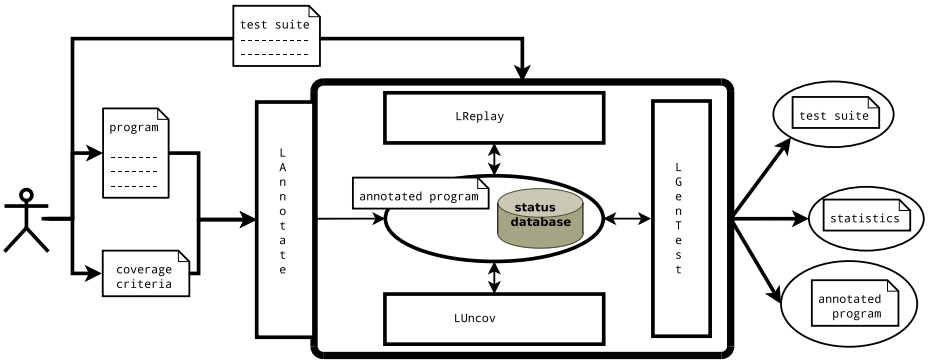

Fig. 4. Overview of LTEST Architecture

3.4 Implementation Details

The LTEST toolkit is built on top of the FRAMA-C verification platform for C programs [4] (open source, LGPL). We took advantage of the plugin-based architecture of FRAMA-C as much as possible, reusing existing analyses of interest for our needs. The FRAMA-C kernel, LANNOTATE, LGENTEST and LUNCOV modules are written in OCaml. The LGENTEST module is based on a modified version of PATHCRAWLER [11], which is written in ECLiPSe/Prolog. The LTEST code is open source (LGPL), except the LGENTEST module, and available online.[1]

4 Experiments

Experiments[1]were conducted to evaluate the interest of the proposed combination of test generation with static analysis and the new optimizations of DSE* in LGENTEST.

[1] Source code, benchmark programs and a detailed description of experiments are available online at `http://micdel.fr/ltest.html`

We consider the same annotated benchmark programs and the same three coverage criteria (CC, MCC and WM) as in [2]. These are standard benchmark programs from the literature, coming from the Siemens test suite, the Verisec benchmark and MediaBench. Their sizes range from a few dozen to a few hundred lines of codes.

We compare the following variants of LGENTEST: the DSE* technique as described in [2], DSE*+s that includes in addition the stopping criterion (**OPT-1**, Sec. 3.3), and DSE*+u+s that exploits in addition uncoverable labels detected through a preliminary pass of LUNCOV (**OPT-2**, Sec. 3.3).

Our results are very promising. First, experiments confirm the interest of the stopping criterion. When full coverage is reached, test generation becomes in average 2.95× faster, and up to 600× faster on some examples. Second, LUNCOV is indeed able to detect several uncoverable objectives, and marks as uncoverable up to 35% of labels in some examples. This yields an improvement of reported coverage ratios by discarding uncoverable objectives. Coverage ratios can thus reach in some cases 100% of coverable objectives. Moreover, by combining the knowledge of statically-detected uncoverable objectives and the stopping criterion, test generation on programs with uncoverable objectives becomes in average 1.36× faster, the speedup going up to 11.52×. Note that the detection of uncoverable objectives takes a reasonable amount of time on our benchmark programs (12% of the total computation time in average, up to 30% on a few cases where test generation terminates quickly, but less than 3% when test generation takes more than 10s). Finally, these experiments underline the real synergy between the two optimisations: the stopping criterion is efficient as long as everything is coverable, while static analysis improves the performances of test generation by removing some uncoverable objectives.

5 Related Work

Many different automatic testing tools are available, from test suite coverage estimation and test replay to automatic test generation. Yet, these tools are usually limited to few services and to few coverage criteria. On the opposite, we aim at providing an integrated and generic toolbox for automated white-box testing.

Most DSE tools support only the basic decision coverage criterion, sometimes enhanced with some implicit "run-time error" coverage criterion (see [2] for a more detailed discussion). An interesting exception is APEX [9], which targets the .NET platform. It operates by adding additional predicates to path conditions during DSE, whereas LTEST annotates the code with predicates. Our approach is less ATG-centric since predicates can be reused outside test generation. In particular, we can use static analysis in order to detect uncoverable labels or measure the (weak) mutation score of a third party's test suite. In addition, LTEST's ATG service implements DSE* [2] dedicated to labels, drastically limiting the overhead observed for example with APEX.

The SANTE approach [3] combines static analysis and DSE in order to prove the absence or presence of run-time errors. The present work can be seen as an extension of SANTE, providing a larger choice of coverage criteria and a larger choice of services (test replay and completion), together with a more flexible combination scheme.

Several recent results have been obtained concerning the combinations of different formal methods inside a single tool [5,6]. FRAMA-C is primarily devoted to

verification rather than testing: plugins collaborate in order to prove assertions written in the high-level language ACSL, cooperation is based on recording which assertions have been proved under which hypotheses [6]. The extensions SANTE and STADY [10] take advantage of dynamic analysis in order to disprove assertions as well. A similar combination has been studied using DAFNY and PEX [5]. While we also combine static and dynamic techniques, our goal is clearly the opposite: we target testing, and use static analysis for optimizing test generation and sharpening coverage measures.

6 Conclusion and Future Work

We propose FRAMA-C/LTEST, a generic and integrated toolkit for automated white-box testing of C programs implemented on top of the FRAMA-C verification platform and relying on a combination of test generation and static analysis.

LTEST can be used in black-box as a powerful testing tool. Yet, thanks to its modular architecture and open-source license, it can also be very useful as a basic building block for developing other advanced testing tools. In the future, we plan to explore additional cooperations with other FRAMA-C plugins and features. For example, we could take advantage of the expressive annotation language ACSL [4] for specifying richer test objectives.

References

1. Ammann, P., Offutt, A.J.: Introduction to software testing. Cambridge University Press (2008)
2. Bardin, S., Kosmatov, N., Cheynier, F.: Efficient Leveraging of Symbolic Execution to Advanced Coverage Criteria. In: ICST 2014. IEEE, Los Alamitos (2014)
3. Chebaro, O., Kosmatov, N., Giorgetti, A., Julliand, J.: Program slicing enhances a verification technique combining static and dynamic analysis. In: SAC 2012. ACM, New York (2012)
4. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C - a software analysis perspective. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) SEFM 2012. LNCS, vol. 7504, pp. 233–247. Springer, Heidelberg (2012)
5. Christakis, M., Müller, P., Wüstholz, V.: Collaborative verification and testing with explicit assumptions. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 132–146. Springer, Heidelberg (2012)
6. Correnson, L., Signoles, J.: Combining Analyses for C Program Verification. In: Stoelinga, M., Pinger, R. (eds.) FMICS 2012. LNCS, vol. 7437, pp. 108–130. Springer, Heidelberg (2012)
7. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed Automated Random Testing. In: PLDI 2005. ACM, New York (2005)
8. Godefroid, P., Levin, M.Y., Molnar, D.: Automated Whitebox Fuzz Testing. In: NDSS 2008 (2008)
9. Jamrozik, K., Fraser, G., Tillman, N., de Halleux, J.: Generating Test Suites with Augmented Dynamic Symbolic Execution. In: Veanes, M., Viganò, L. (eds.) TAP 2013. LNCS, vol. 7942, pp. 152–167. Springer, Heidelberg (2013)
10. Petiot, G., Kosmatov, N., Giorgetti, A., Julliand, J.: How Test Generation Helps Software Specification and Deductive Verification in Frama-C. In: Seidl, M., Tillmann, N. (eds.) TAP 2014. LNCS, vol. 8570, pp. 204–211. Springer, Heidelberg (2014)
11. Williams, N., Marre, B., Mouy, P.: On-the-Fly Generation of K-Path Tests for C Functions. In: ASE 2004. IEEE, Los Alamitos (2004)