Taming Coverage Criteria Heterogeneity with LTest*

Michaël Marcozzi, Sébastien Bardin, Mickaël Delahaye, Nikolai Kosmatov, Virgile Prevosto
CEA, LIST, Software Reliability Laboratory
91191 Gif-sur-Yvette, France
firstname.lastname@cea.fr

***Abstract*— Automated white-box testing is a major issue in software engineering. In previous work, we introduced LTest, a generic and integrated toolkit for automated white-box testing of C programs. LTest supports a broad class of coverage criteria in a unified way (through the *label* specification mechanism) and covers most major parts of the testing process – including coverage measurement, test generation and detection of infeasible test objectives. However, the original version of LTest was unable to handle several major classes of coverage criteria, such as MCDC or dataflow criteria. Moreover, its practical applicability remained barely assessed.**

In this work, we present a significantly extended version of LTest that supports almost all existing testing criteria, including MCDC and some software security properties, through a native support of recently proposed *hyperlabels*. We also provide a more realistic view on the practical applicability of the extended tool, with experiments assessing its efficiency and scalability on real-world programs.

I. Introduction

Context. Automated white-box testing is a major topic in software engineering [1], [2], [3], [4]. Along the years, many tools have been proposed for supporting different parts of the testing process. These tools explicitly or implicitly rely on a *code-coverage criterion* (a.k.a. *adequacy criterion* or *testing criterion*) [3], [4] to guide automation. Such a criterion formally specifies what the *test objectives* are. These can then be used to assess the quality of a *test suite* and to guide the selection of additional test cases. In previous work [5], Bardin et al. introduced LTest, a *generic* and *integrated* toolkit for automated white-box testing of C programs. LTest is generic in the sense that it handles a wide set of coverage criteria in a unified way. It is also integrated in the sense that it centralizes heterogeneous techniques to automatize most key tasks in white-box testing. Indeed, in addition to test replay and coverage measurement, the tool leverages a dedicated version of Dynamic Symbolic Execution [6], [7] for providing coverage-oriented test generation [8]. It also relies on static analyses from the Frama-C [9] platform to provide efficient detection of uncoverable test objectives [10].

Goals and Contributions. While the original version of LTest already supported a large scope of criteria, it relied on a specification mechanism whose expressiveness remained limited with regard to some other exiting criteria. As a consequence, LTest was unable to handle several classes of criteria such as strong variants of MCDC, as well as criteria based on data-flow analysis or path exploration. Yet, such criteria can be very important in practice. In particular, MCDC coverage is mandated by the DO-178 standard that dictates the development process of avionics software. On the other hand, the practical applicability of the tool remained barely illustrated and assessed, as [5] only reported preliminary results, on small-scale benchmarks. The goals of the present work are (1) to enable a better support of (almost) all existing criteria in LTest, and (2) to provide a more realistic view on its practical applicability, by studying its efficiency and scalability on real-world code.

Test automation in LTest relies on annotating the tested code with the considered test objectives using a generic (i.e. criterion independent) test objective specification language: labels [8]. The limitations of this language prevent LTest from handling criteria like MCDC or dataflow criteria. Very recently, we have provided a conceptual extension of the label language, called HTOL [11], that overcomes previous limitations of labels and allows for encoding almost all criteria from the literature (except strong mutations). HTOL can also be used to test important software security properties. An additional goal of this work is to provide a tool support for the HTOL language.

- As a first contribution of this paper, we report on significant advances made in an extended version of LTest that now offers a support for HTOL test objectives, and detail the new technical capabilities of some of its modules. We show how these new features can be exploited in practice, by illustrating how one can use the new LTest API to add support for new testing criteria.
- As a second contribution, we perform an experimental study of efficiency and scalability of the new capabilities of LTest. The experiments involve coverage measurement and test generation. We consider test suites up to 10,000 tests and perform unit testing on C functions from real world programs, including OpenSSL and SQLite.

These contributions make LTest a practical, universal and extensible white-box testing toolkit, which is now released with built-in support for 14 major coverage criteria. LTest users can benefit from advanced techniques for automating their practical testing tasks, whatever the approach they choose for estimating coverage. Developers of new test automation techniques can build them directly inside LTest, making them immediately available in practice, no matter the way coverage is defined.

Outline. Section II gives an overview of the original version of LTest. Section III provides a practical presentation of HTOL,

* Work partially funded by French ANR (grant ANR-12-INSE-0002).

defined in [11]. Section IV details the new technical capabilities of LTest, lifted to most existing test criteria. Section V discusses efficiency and scalability experiments. Finally, related work and conclusion are discussed in Sections VI and VII.

II. Original version of LTest

A. Main Features

Given a C program to be tested according to the test objectives defined by a code coverage criterion, the LTest toolkit [5] offers the following services:

Uncoverability detection tries to detect which of the test objectives cannot be covered by any test datum (e.g. in dead code). Its results are primarily used by the other two services, but can also be exported for external use.

Coverage measurement replays an existing test suite and reports which of the test objectives have been covered, which have not and which are uncoverable.

Test generation creates a test suite tailored to cover as many test objectives as possible. It can skip objectives known to be uncoverable, or those already covered by a given test suite in order to complete its coverage.

The following criteria are supported and can be combined with each other: 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). The analysis can be restricted to certain functions in the code and additional test objectives can be added manually in the code.

B. Specifying Test Objectives with Labels

The toolkit has been conceptually designed around the notions of *labels* [8] and *annotated programs*, which provide a specification mechanism for coverage criteria. *Labels* are predicates attached to program statements. A program with labels is called an annotated program. A label is covered if it is reached by a test case execution and its predicate is satisfied. Labels can simulate many common coverage criteria, from decision or condition coverage to a substantial subset of weak mutations, making it possible to handle them all in a unified way. For each test objective defined by the criterion, a label is added to the program under test, such that covering the label is equivalent to covering the objective. The automatic insertion of adequate labels for a given coverage criterion is performed by a so-called *labelling function*. An example is given in Fig. 1.

```
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;
```

Fig. 1. Simulating decision coverage (DC) criterion with labels

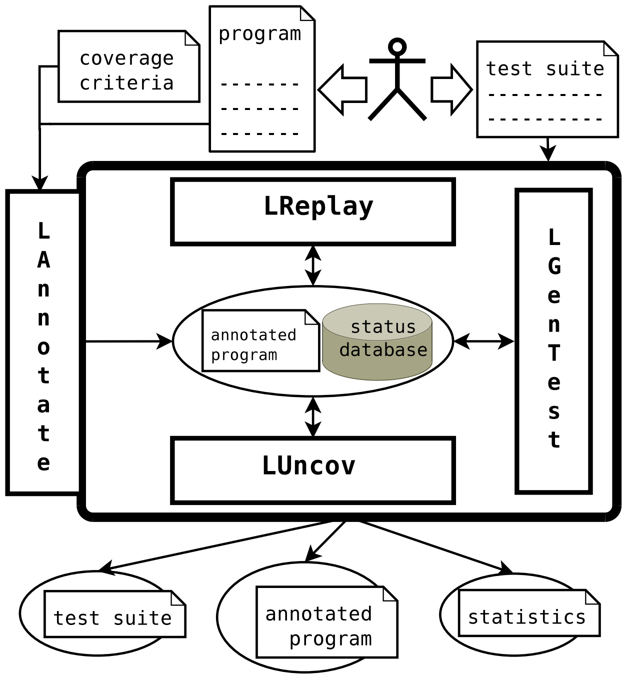

Fig. 2. Overview of LTest Architecture

C. Internal Architecture

LTest comes as a series of four plugins of the Frama-C [9] platform, mostly written in OCaml: LAnnotate, LReplay, LUncov and LGenTest. These modules interact through shared information made of the **annotated program** and a **status database** mapping each label to its current status: *covered*, *uncoverable* or *uncovered*. The whole architecture is depicted in Figure 2. We provide hereafter the main clues about the role of each module. The LTest code is open source (LGPL), except the LGenTest module, and available online[1].

LAnnotate acts as a front-end: it annotates the program with labels according to the chosen criteria and creates the status database. The module implements the idea of labelling functions and provides one for each supported criterion. In addition, users can extend the module by writing their own labelling functions. To facilitate this task, LAnnotate provides an API to easily insert labels into the code and to register inserted labels in the shared status database.

Given an annotated program and its status database, the **LUncov** module runs static analysis to identify uncoverable labels and mark them as uncoverable in the database [10].

Provided with a test suite and an annotated program, the **LReplay** module executes each test case in order to update the label statuses in the status database. In addition, it computes coverage statistics for the given test suite.

The **LGenTest** module provides the test generation service of LTest, by implementing a flavour of Dynamic Symbolic Execution (DSE) [6], [7] tailored to cover labels and called DSE* [8]. The tool is based on a modified version of the

[1]http://micdel.fr/ltest.html

PathCrawler test generator [7]. LGenTest reads the status database so that already-covered labels (i.e. by another test suite) and uncoverable labels (detected by LUncov) are ignored. LGenTest updates the status database recording the labels newly covered by the tests it has generated.

III. A Practical Introduction to HTOL

We recently introduced HTOL [11], a major extension of the label language used in LTest. HTOL makes it possible to emulate all white-box criteria defined in the Ammann and Offutt's book [4], except strong mutations. Compared to labels, HTOL notably adds support for all the variants of the MCDC criterion, call coverage and all the dataflow and path-based criteria. Moreover, HTOL enables encoding test objectives to find violations of important software security properties, such as non-interference [12].

Concretely, HTOL introduces five operators to combine (now referred to as *atomic*) labels into *hyperlabels*, specifying more complex test objectives:

Bindings $\triangleright$ save the value of well-defined expression(s) at the state at which an atomic label is covered;
Guard $\langle\ \mid \cdot\rangle$ expresses a constraint over the binding values of several atomic labels;
Sequence $\xrightarrow{\cdot}$ requires atomic labels to be covered sequentially by a single constrained test run;
Conjunction $\cdot$ and **Disjunction** $+$ are the logical AND and OR operators, for combining test objectives together.

We illustrate briefly these operators through two examples of criteria encoding.

Example 1 (MCDC) We start with conjunction, bindings and guards. Consider the following code snippet:

```
statement_0;
// loc_1
if (x==y && a<b) {...};
statement_2;
```

The (strong) **MCDC** criterion requires to demonstrate here that each atomic condition $c_1 \triangleq$ x==y and $c_2 \triangleq$a<b alone can influence the whole branch decision $d \triangleq c_1 \wedge c_2$. For c_1, it comes down to providing two tests where the truth value of c_2 at loc_1 remains the same, while values of c_1 and d change. The requirement for c_2 is symmetric. This can be directly encoded with hyperlabels h_1 and h_2 as follows:

$$l \triangleq (loc_1, \textit{true}) \triangleright \{c_1 \hookleftarrow \texttt{x==y};\ c_2 \hookleftarrow \texttt{a<b};\ d \hookleftarrow \texttt{x==y\&\&a<b}\}$$
$$l' \triangleq (loc_1, \textit{true}) \triangleright \{c_1' \hookleftarrow \texttt{x==y};\ c_2' \hookleftarrow \texttt{a<b};\ d' \hookleftarrow \texttt{x==y\&\&a<b}\}$$
$$h_1 \triangleq \langle l \cdot l' \mid c_1 \neq c_1' \wedge c_2 = c_2' \wedge d \neq d' \rangle$$
$$h_2 \triangleq \langle l \cdot l' \mid c_1 = c_1' \wedge c_2 \neq c_2' \wedge d \neq d' \rangle$$

h_1 requires that the test suite reaches loc_1 twice (through the $\cdot$ operator between labels l and l'), with one or two tests The values taken by the atomic conditions and the decision when loc_1 is reached are bound (through $\triangleright$) to metavariables c_1, c_2, d for the first execution and to c_1', c_2', d' for the second one. Moreover, these recorded values must satisfy the guard $c_1 \neq c_1' \wedge c_2 = c_2' \wedge d \neq d'$, meaning that c_1 alone can influence the decision. Similarly, h_2 ensures the desired test objective for c_2.

Example 2 (all-defs) We illustrate now the sequence and disjunction operators. Consider the following code snippet.

```
/* loc_1 */ a := x;
if (...) /* loc_2 */ res := x+1;
else /* loc_3 */ res := x-1;
```

In order to meet the **all-defs** dataflow criterion for the definition of variable a at line loc_1, a test suite must cover at least one of the two def-use paths from loc_1 to loc_2 and to loc_3. This objective is represented by the hyperlabel $h \triangleq ((loc_1, \textit{true}) \rightarrow (loc_2, \textit{true})) + ((loc_1, \textit{true}) \rightarrow (loc_3, \textit{true}))$.

IV. Upgrading LTest for Wider Criterion Support

We describe in this section how we have upgraded LTest in order to support the whole HTOL specification language, lifting LTest into an (almost) universal testing tool. Such an upgrade requires major changes in the LAnnotate and LReplay modules, to be able to instrument code with hyperlabels and to measure hyperlabel coverage (as a function of label coverage and coverability). Updates to the LGenTest and LUncov modules are less necessary, as they can already be used (in a suboptimal way) to find tests that will cover the atomic labels composing the hyperlabels and to detect that some of these atomic labels (hence the hyperlabels that are built upon them) are uncoverable.

Experiments evaluating the upgrade of LTest are provided in Section V. *The source code of the upgraded LAnnotate and LReplay modules is available on the companion website*[2].

A. Upgrading LAnnotate: Instrumenting Code with HTOL

Code annotation and status database format. As a first step of the upgrade process for LAnnotate, we redesigned the output format that it uses for annotating the code with (hyper)labels and generating the (hyper)label status database. Figures 3 and 4 illustrate this new format, considering the same examples as in Section III. From a given C source file, LAnnotate produces two resulting files:

- a C file with the annotated code, containing specific macros at the places where an atomic label should be put, and which are no-ops by default.
- an `.hyperlabel` file, which represents the initial state of the status database, where no hyperlabel has been covered, hence no binding is associated to any label.

We detail in section IV-B how LReplay uses these files and updates the information contained in the `.hyperlabel` file when running a test suite.

There can basically be two sets of macros in the annotated C file. First, `label` corresponds to atomic labels and defines the label predicate (1 for *true* in C), its identifying number as well as the number, name and symbolic value of the metavariables. Second, `seq_label` and `seq_cond` correspond to HTOL's

[2]Companion website: http://icst17.marcozzi.net

```
int func
(int x, int y,
 int a, int b) {
if (x==y && a<b)
 return a;
else
 return b;
}
```

file.c

$\longrightarrow$

```
#ifndef label
#define label(...) do{}while(0)
#endif
int func
(int x, int y,
 int a, int b) {
label(1,1,3,"c1",x==y,
      "c2",a<b,"d",x==y && a<b);
label(1,2,3,"c1p",x==y,
      "c2p",a<b,"dp",x==y && a<b);
if (x==y && a<b)
 return a;
else
 return b; }
```

file_annotated.c

+

```
// Labels
// id, coverage, nb_bindings, bindings
1,notcov,3,c1,,c2,,d,,
2,notcov,3,c1p,,c2p,,dp,,
// Sequences
// Hyperlabels
// id, hyperlabel, coverage
1,<l1.l2|c1!=c1p && c2==c2p && d!=dp>,notcov
2,<l1.l2|c1==c1p && c2!=c2p && d!=dp>,notcov
```

file.hyperlabel

Fig. 3. Files produced by LAnnotate when applied on a C file for the strong MCDC criterion

```
void fun() {
int x = read_data();
if (read_cond())
 return x+1;
else
 return x-1;
}
```

file.c

$\longrightarrow$

```
#ifndef seq_label
#define seq_label(...) do{}while(0)
#endif
void fun() {
seq_label(1,1,1,"tag",0);
seq_label(1,2,1,"tag",0);
int x = read_data();
if (read_cond()) {
 seq_label(1,1,2,"tag",0);
 return x+1;
} else {
 seq_label(1,2,2,"tag",0);
 return x-1; }
```

file_annotated.c

+

```
// Labels
// Sequences
// id, coverage, nb_bindings, bindings
1,notcov,0,
2,notcov,0,
// Hyperlabels
// id, hyperlabel, coverage
1,s1+s2,notcov
```

file.hyperlabel

Fig. 4. Files produced by LAnnotate when applied on a C file for the all-defs criterion

sequences. Indeed, it should be noted that the syntax and semantics of HTOL enables one to constrain the sequence operator, i.e. to place a constraint on several steps of the execution paths. This forces us to monitor what happens during several consecutive statements. For example, the hyperlabel $(loc_1, true) \xrightarrow{loc_2 \Rightarrow !(a>0)} (loc_3, true)$ should be covered by an execution traversing loc_1 and loc_3 without entering the loop at loc_2 in the following code:

```
/* loc_1 */ statement_1
/* loc_2 */ while (a > 0) { ... }
/* loc_3 */ statement_3
```

This is handled by LAnnotate in the following way:

```
#ifndef seq_cond
#define seq_label(...) do{}while(0)
#define seq_cond(...) do{}while(0)
#endif
...
seq_label(1,1,1,"seq_section1",0);
/* loc_1 */ statement_1
seq_cond(!(a>0),"seq_section1");
/* loc_2 */ while (a > 0) { ... }
seq_label(1,1,2,"",0);
/* loc_3 */ statement_3
```

`seq_label` corresponds to the atomic labels composing the sequences and defines the label predicate, an identifying number for the sequence, the rank occupied by the label in this sequence, a tag name for the sequence section following the label, as well as the number, name and symbolic value of the metavariables. `seq_cond` corresponds to sequence conditions and defines the symbolic value of the condition as well as the tag name of the (constrained) sequence section. A single `seq_cond` can thus constrain several sequence sections if they share the same tag name. This is useful as many sequence sections are often constrained by the same condition in practice.

API for developing hyperlabelling functions. As a second step of the upgrade process for LAnnotate, we have redesigned its API so that it can be used to create and store *hyperlabeling functions*. Hyperlabelling functions are pieces of code dedicated to the handling of a particular coverage criterion. Once developed, they can be plugged into LAnnotate to provide support for additional criteria. Practically, an hyperlabelling function is responsible, for a given C source file received as input, to generate the corresponding macro-annotated and `.hyperlabel` files, populated with the test objectives defined by the considered criterion.

The main primitives offered by the new LAnnotate API for developing and registering hyperlabelling functions are presented hereafter:

Registration Service

– Register a new hyperlabelling function.
– Fetch an hyperlabelling function by criterion name.

Code Annotation Service
(notably relying on the API of the Frama-C platform)

– Parse a C file into a standardized abstract syntax tree (AST) and output such an AST as code in a file.
– Extract useful information out of some specific AST nodes, using a visitor pattern.
– Extract the control-flow graph (CFG) from the AST of a C file. Navigate this CFG and obtain syntactic and semantic information about the navigated code. These primitives are particularly useful for generating adequate hyperlabels for criteria specifying complex

set of paths to be covered in the CFG.
– Insert a dedicated macro node for label and sequence annotation before or after some specific AST nodes, using a visitor pattern.

Status Database Creation Service
– Save a set of label definitions as a CSV file.
– Save a set of hyperlabel definitions into a file, following the syntax of a LALR(1) grammar for HTOL.

Built-in hyperlabelling functions. The labelling functions built in the original version LTest were adapted to this new API, while new hyperbelling functions were developed for some important criteria not encodable with labels. The current version of LAnnotate is released with 14 built-in hyperllabeling functions, for the following criteria: strong, masking and weakened MCDC, all-definitions and all-uses coverage, weak mutations, function and call coverage, condition, decision, multiple condition and n-wise condition coverage, general inactive clause coverage, input domain partition coverage.

B. Upgrading LReplay: measuring hyperlabel coverage

In this section, we show how LReplay has been redesigned so that it can now play a given test suite and detect which hyperlabels are covered.

Test harness. LTest provides a generic testing infrastructure for playing a test suite and collecting test data. Each test case must be encoded into a normalized test driver, i.e. a C file responsible for preparing the test, calling the tested code, saving the result and killing possibly looping tests (with a time-out). Test suites generated by LGenTest are automatically encoded into such test drivers. Once the tested code has been annotated with labels and sequences by LAnnotate, LReplay can compile and run it with the test driver and some instrumentation code. This last code is a set of C macro definitions of the form

```
#define label(/* some parameters */)
                    call_lreplay(/* same parameters */)
```

which overwrite the label and sequence annotations in the tested code so that, every time the test execution reaches an annotation, the corresponding coverage info is sent to LReplay.

Computing hyperlabel coverage. Given the *file_annotated.c* and *file.hyperlabel* files produced by LAnnotate as inputs, as well as a set of test drivers, LReplay follows a three step procedure:

normalization First, *file.hyperlabel* is parsed and each hyperlabel is transformed into its *disjunctive normal form* (DNF) [11].

harvesting Second, test drivers are compiled with the instrumented code and run. Every atomic label and label sequence covered during a test run is marked on-the-fly in the *file.hyperlabel* file. LReplay also saves the environment (values of metavariables) that instantiates the label bindings at the coverage points.

consolidation Third, the label and sequence coverage information is propagated within every hyperlabel from *file.hyperlabel*, in order to establish if the test suite covers it or not. Hyperlabel coverage data is saved in *file.hyperlabel* as well.

These steps are described in more details in the following.

Normalization. As stated in [11], any hyperlabel can be rewritten into an equivalent hyperlabel in DNF. Each test objective is thus encoded as a disjunction of guarded conjunctions between atomic labels and sequences. This form of hyperlabels is both very convenient for coverage measurement and very common in practice. In LReplay, the transformation is performed on the fly, while parsing the *file.hyperlabel* file, using an attribute grammar. *Details about the transformation algorithm can be found on the companion website.*

Environment harvesting. Once hyperlabels in DNF have been obtained, each test driver from the suite is run, and the coverage information for basic labels, sequences and binding values is collected in memory. Note that we need to store all possible binding values encountered along the execution of a test, not just the first one. While this is easy for atomic labels, sequences must be treated with care, as there are some non deterministic choices there. Namely, if the sequence includes a loop, its starting point may occur at any of the n^{th} loop steps, and LReplay must maintain a whole set of partially executed sequences before being able to choose the appropriate one. Due to space limitations, we do not describe this point further, as it is common in runtime monitoring. *A detailed description is nevertheless available on the companion website.*

Consolidating coverage result. Once the coverage information for basic labels, sequences and binding values is fully collected, we can compute the whole hyperlabel-coverage information. This is straightforward on DNF hyperlabels:

- atomic labels and sequences with no guard are covered iff they have been covered in the harvesting step;
- a guarded conjunction is covered iff each of its label or sequence has been marked as covered during harvesting and there is at least one valid combination of the collected binding values such that the guard condition is true;
- a disjunction is covered iff at least one of its components is covered.

In practice, the tool tries every possible valid combination of the collected binding values for every guarded conjunction, until it finds one which makes the guard condition true or proves that none exists (in which case the hyperlabel is not covered by the test suite).

Optimizations. We first preprocess hyperlabels under consideration in order to remove all unused metavariables appearing in bindings. Then, during harvesting, we ensure that each binding is recorded only once, avoiding duplicated values. Finally, we perform conjunction and disjunction evaluation in a lazy way, in order to avoid unnecessary combinatorial reasoning on guarded conjunctions.

About complexity. The procedure presented so far runs in worst-case exponential time, mainly because of three factors: (1) normalization may yield an exponential-size hyperlabel, (2) consolidation for guarded conjunctions may lead to checking a number of solutions exponential in the size of the conjunction, and (3) monitoring sequences of labels may include harvesting a number of environments exponential in the length of the considered run.

Yet, in practice, LReplay appears to *perform well on existing classes of testing criteria* (cf. Section V-A). Here are a few explanations. First, common testing criteria are naturally in DNF. Second, the critical parameters indicated above are strongly limitated in existing criteria: conjunctions of length 2; sequences of length 2 or without bindings; small domains of metavariables (boolean). In that setting, complexity becomes polynomial.

C. Exploiting LGenTest and LUncov Capabilities with HTOL

LGenTest provides a test generation mechanism aimed at covering test objectives encoded as labels in the tested code. However, in HTOL, the test objectives are encoded using a more refined process. A set of secondary objectives are first encoded by original labels. The primary objectives are then obtained by further constraining these secondary objectives, using the HTOL operators. In order to generate tests for HTOL-encoded objectives, LGenTest can be used, with no change, on the secondary objectives. Such an approach is suboptimal, as tests covering the secondary objectives do not necessarily satisfy the primary ones. However, generating tests in this way remains relevant, as tests covering the primary objectives must always first satisfy the secondary ones. As an example, the **GACC** variant of the **MCDC** criterion can be encoded with labels only, while the **CACC** and **RACC** variants require constraining the same labels with conjunctions and guards. The test suites generated by LGenTest for **GACC** (or even for a weaker but less demanding criterion such as **CC**) can be naturally used for trying to cover **CACC** and **RACC**. We provide experiments evaluating this approach in Section V-B.

LUncov is able to detect labels that are not coverable, as they may be part of a dead portion of code or because their predicate might be infeasible. Such information can be directly propagated by LReplay to prune out uncoverable hyperlabels, using the following rules: a binding is uncoverable if the bound label is uncoverable, a sequence is uncoverable if at least one label is uncoverable, a (guarded) conjunction is uncoverable if at least one conjunct is uncoverable, a (guarded) disjunction is uncoverable if all the disjuncts are uncoverable. Again, such an approach is suboptimal, as several cases of hyperlabel uncoverability will never be detected in this way, such as infeasible guard or sequence predicates.

V. Experimental Evaluation

A. Assessing LAnnotate and LReplay

Objective. In this section, we want to assess the efficiency of the newly redesigned LAnnotate and LReplay modules for unit testing. More precisely, we seek to answer the following research question: how do LAnnotate and LReplay scale with large test suites on criteria using hyperlabel operators?

Protocol. We consider 13 C functions split up into 3 groups:

- 5 functions, mainly from Siemens [13], Verisec [14] & MediaBench [15], as already used in [8];
- 5 functions from OpenSSL 1.0.2 [16], a 250 $kloc$ open-source application. We focus on modules of about 1 $kloc$.
- 3 functions from SQLite 3.13 [17], a 215 $kloc$ open-source application. We focus on modules of a few $kloc$.

The C files automatically annotated with HTOL test objectives are available on the companion website.

A set of up to 10,000 test cases is randomly generated for each C function. Our tool is successively run with an increasing number of these unit test cases, also available *from the companion website*. Each tool run is repeated 7 times. First, tests are executed without measurement (baseline). Then, we measure coverage for the **CC** and **GACC** criteria encodable in labels (used as a witness). Finally, we measure coverage for the **CACC**, **RACC**, **FCC** and **all-defs** criteria, which involve the five operators from hyperlabels. All experiments are performed under Ubuntu Linux 14.04 on an Intel Core i7-4712HQ CPU at 2.30GHz, with 16GB of RAM.

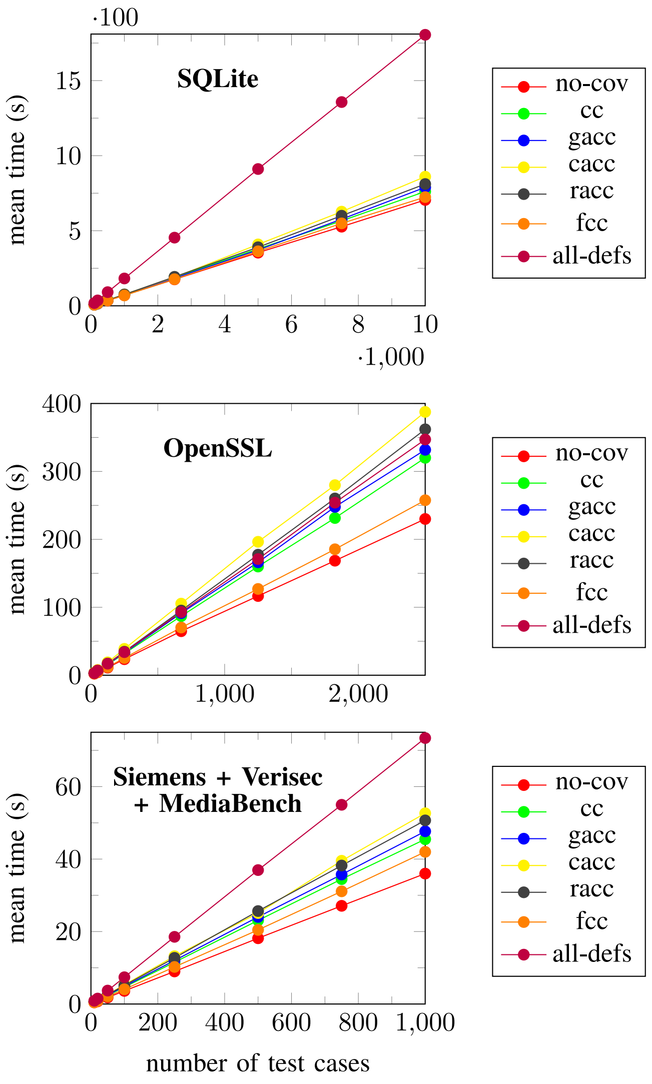

Fig. 5. Scalability of Coverage Measurement

Results and discussion. *Detailed results are on the companion website*. Figure 5 plots, for each criterion and the baseline (no-cov), the mean measurement time for all programs, as a function of the test suite size. We can notice that: (1) the measurement time grows linearly with the number of test cases, (2) the time overhead is very reasonable for all criteria but **all-defs** (between 1.1x and 2x), and still not so high for **all-defs** (between 2x and 4x), and (3) these results hold on the three benchmarks, regardless of program size. Note that **all-defs** yields a tangible time overhead on some programs, due to the higher number of test objectives that are defined. However, many objectives are trivial or redundant, which could be detected using some static analysis techniques in an optimized version of the tool.

Conclusion. These results indicate that upgraded versions of LAnnotate and LReplay provide both practically applicable and (almost) criterion-independent testing capabilities. The measurement time for criteria beyond labels is acceptable and remains linear with the size of the test suite. Moreover, as our tool implementation can be further optimized, there is still room for a strong reduction of coverage measurement time, when using the approach in an industrial context.

B. *Assessing LGenTest*

Objective. We want to assess whether LGenTest can be efficient even for criteria beyond labels. More precisely, we address in this section the following research question: can the label-directed generation of LGenTest provide sufficient coverage for criteria that cannot be encoded with labels only?

Protocol. We consider here two versions of the **MCDC** criterion non encodable in labels: **CACC** and **RACC**. For each of the 5 functions in the first group defined at Section V-A, LGenTest is first called to produce test suites for the **CC**, **MCC** and **GACC** criteria encodable in labels. LReplay is then used to measure the coverage achieved for the criteria **CACC** and **RACC** by these test suites as well as by witness test suites built by random test generation.

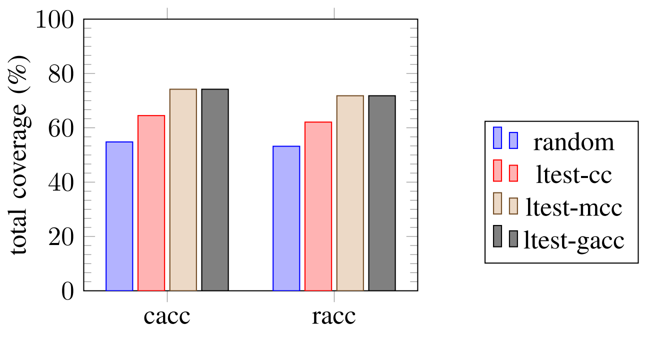

Fig. 6. Coverage of Label-Directed vs. Random Test Generation for CACC and RACC criteria

Results and discussion. Figure 6 shows the total coverage. Label-directed generation provides a significantly better coverage level than random generation. This is particularly the case with **MCC** and **GACC** based generation, as those criteria are stronger than **CC**. The **RACC**-coverage obtained on a program with **GACC**-directed generation ranges between 61% and 94%, with an average of 72%, while for random testing, this number ranges between 29% and 81%, with an average of 53%.

Conclusion. Combining the upgraded versions of LAnnotate and LReplay with LGenTest provides a full-featured testing tool, that handles both coverage measurement and test generation. Our initial experiments indicate that automatic test generation for HTOL is feasible with LGenTest (at least for small programs): LTest provides an acceptable coverage ratio for criteria beyond labels (on average 19% better than random testing on tested programs for **CACC** and **RACC** criteria).

VI. Related Work

A large number of automatic testing tools are available. However, they often offer a limited scope of services (test coverage measurement or automatic test generation) and are restricted to few coverage criteria. On the contrary, LTest is an integrated and generic toolkit for automated white-box testing.

Coverage measurement tools. Code coverage is used extensively in the industry. As a result, there exists a lot of tools that embed some sort of coverage measurement. For instance, in 2007, a survey [18] found ten tools for programs written in the C language: Bullseye [19], CodeTEST, Dynamic [20], eXVantage, Gcov (part of GCC) [21], Intel Code Coverage Tool [22], Parasoft [23], Rational PurifyPlus [24], Semantic Designs [25], TCAT [26]. To this date, there are even more tools, such as COVTOOL [27], LDRAcover [28], and Testwell CTC++ [29]. As a rule of thumb, these tools support a limited number of test criteria in a hard-coded, non-generic manner. Table I summarizes implemented criteria for some popular tools. However, to be fair, these code coverage tools also aim at causing as little overhead as possible. In contrast, as a first step, we only aim at getting a reasonable overhead.

Test generation tools. Many test generation tools only handle basic coverage criteria, such as bounded path coverage or decision coverage. Three interesting exceptions to be compared with our work are Fshell [30], FAJITA [31] and Apex [32].

Fshell enables encoding code coverage criteria into an extended form of regular expressions, whose alphabet is composed of elements from the control-flow graph of the tested program. It then takes advantage of an off-the-shelf model-checker to automatically generate a test suite satisfying such a specification. The scope of criteria that can be encoded in FShell is incomparable with the one offered by LTest, but the tool cannot encode neither **MCDC** nor weak mutations.

The FAJITA tool proposes to encode a testing criterion as a set of disjoint boolean constraints, which partition the input space of the tested program. These constraints are then solved using a SAT solver to build a test suite satisfying the criterion. The paper demonstrates how basic white-box criteria (statement, branch and path coverage) can be handled in this way.

As our tool, Apex also relies on dynamic symbolic execution, but adds additional predicates to the path conditions, where LTest annotates the code itself. However, LTest makes use of

Tool name	BBC	FC	DC	CC	DCC	GACC	MCDC	MCC	BP	Other
Gcov	✓	✓	✓							0/19
Bullseye		✓			✓					0/19
Parasoft	✓	✓	✓	✓			✓		✓	0/19
Semantic Designs		✓	✓							0/19
Testwell CTC++	✓	✓			✓		✓			0/19
Original LTest [5]	✓	✓	✓	✓	✓	✓		✓		4/19
Extended LTest	✓	✓	✓	✓	✓	✓	✓	✓	✓	18/19

TABLE I
SUPPORT OF WHITE-BOX COVERAGE CRITERIA [4] IN COVERAGE MEASUREMENT TOOLS

a DSE tailored to labels, which strongly limits the overhead observed with Apex.

VII. CONCLUSION

LTest has been briefly introduced two years ago as a toolkit for white-box testing. It could have been presented as a coherent combination of various advanced techniques within a testing tool, able to automate most aspects of testing, for various classes of coverage criteria. In this work, we have described a major conceptual upgrade of the core design of LTest and provided a deeper insight into its practical applicability.

The extended capabilities of the code instrumentation and coverage measurement modules make LTest able to handle all existing white-box criteria but strong mutations, and to do it in a generic way. Any technique for test automation built in LTest (like DSE or static analysis) is thus immediately available for all criteria. At the same time, genericness makes it easy to add support for new criteria (such as test objectives detecting violations of common security properties). Any additional criterion will also immediately benefit for free from all test automation features of LTest.

This paper also provides a bunch of useful information on how LTest is actually implemented, together with its last source code. Furthermore, it details several experiments showing the scalability and efficiency of the tool on real-world code. These elements constitute a strong hint at the practical applicability of the tool, both for developers and users.

In future work, we intend to push the genericness and practical applicability of the tool even further. First, by upgrading LGenTest and LAnnotate for optimal test generation and uncoverability detection with HTOL. Second, by implementing support for security property testing. Finally, by developing the emerging interest for the tool in the industrial world.

REFERENCES

[1] J. Myers, Glenford, C. Sandler, and T. Badgett, *The Art of Software Testing*, 3rd ed. Wiley, 2011.
[2] P. Mathur, Aditya, *Foundations of Software Testing*. Addison-Wesley Professional, 2008.
[3] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Comput. Surv.*, 1997.
[4] P. Ammann and J. Offutt, *Introduction to Software Testing*, 1st ed. Cambridge University Press, 2008.
[5] S. Bardin, O. Chebaro, M. Delahaye, and N. Kosmatov, "An all-in-one toolkit for automated white-box testing," in *TAP*. Springer, 2014.
[6] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *PLDI*. ACM, 2005.
[7] N. Williams, B. Marre, and P. Mouy, "On-the-fly generation of k-paths tests for C functions: towards the automation of grey-box testing," in *ASE*. IEEE, 2004.
[8] S. Bardin, N. Kosmatov, and F. Cheynier, "Efficient leveraging of symbolic execution to advanced coverage criteria," in *ICST*. IEEE, 2014.
[9] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-C: A program analysis perspective," *Formal Asp. Comput.*, 2015.
[10] S. Bardin, M. Delahaye, R. David, N. Kosmatov, M. Papadakis, Y. Le Traon, and J.-Y. Marion, "Sound and quasi-complete detection of infeasible test requirements," in *ICST*. IEEE, 2015.
[11] M. Marcozzi, M. Delahaye, S. Bardin, N. Kosmatov, and V. Prevosto, "Generic and effective specification of structural test objectives," in *ICST*. IEEE, 2017.
[12] M. R. Clarkson and F. B. Schneider, "Hyperproperties," *J. Comput. Secur.*, 2010.
[13] H. Do, S. Elbaum, and G. Rothermel, "Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and Its Potential Impact," *Empirical Software Engineering*, 2005.
[14] K. Ku, T. E. Hart, M. Chechik, and D. Lie, "A Buffer Overflow Benchmark for Software Model Checkers," in *ASE*. ACM, 2007.
[15] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: a tool for evaluating and synthesizing multimedia and communications systems," in *ACM Int. Symp. on Microarchitecture*, 1997.
[16] "OpenSSL," https://www.openssl.org.
[17] "SQLite," https://www.sqlite.org.
[18] Q. Yang, J. J. Li, and D. M. Weiss, "A survey of coverage-based testing tools," *The Computer Journal*, 2009.
[19] "Bullseye Testing Technology: BullseyeCoverage," http://bullseye.com/.
[20] "Dynamic Code Coverage," http://dynamic-memory.com/.
[21] "GCC's Gcov," https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.
[22] "Intel Code Coverage Tool in Intel C++ compiler," https://software.intel.com/en-us/node/512810.
[23] "Parasoft C/C++test: Comprehensive dev. testing tool for C/C++," https://www.parasoft.com/product/cpptest/.
[24] "PurifyPlus: Run-Time Analysis Tools for Application Reliability and Performance," http://teamblue.unicomsi.com/products/purifyplus/.
[25] "Semantic designs: C test coverage tool," http://semanticdesigns.com/Products/TestCoverage/CTestCoverage.htm.
[26] "TCAT," http://www.testworks.com/Products/Coverage/tcat.html.
[27] "COVTOOL - Free test coverage analyzer for C++," http://covtool.sourceforge.net/.
[28] "LDRA – LDRACover," http://www.ldra.com/en/ldracover.
[29] "Testwell CTC++: Test coverage analyzer for C/C++," http://www.testwell.fi/ctcdesc.html.
[30] A. Holzer, C. Schallhart, M. Tautschnig, and H. Veith, "Fshell: Systematic test case generation for dynamic analysis and measurement," in *CAV*. Springer, 2008.
[31] P. Abad, N. Aguirre, V. Bengolea, D. Ciolek, M. F. Frias, J. Galeotti, T. Maibaum, M. Moscato, N. Rosner, and I. Vissani, "Improving test generation under rich contracts by tight bounds and incremental sat solving," in *ICST*. IEEE, 2013.
[32] K. Jamrozik, G. Fraser, N. Tillman, and J. de Halleux, "Generating test suites with augmented dynamic symbolic execution," in *TAP*. Springer, 2013.