

# Generic and Effective Specification of Structural Test Objectives

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

*Abstract*—While a wide range of different and sometimes heterogeneous code-coverage criteria (a.k.a. testing criteria, or adequacy criteria) have been proposed, there exists no generic formalism to describe them all, and available test automation tools usually support only a small subset of them. We introduce a new specification language, called HTOL (Hyperlabel Test Objectives Language), providing a powerful generic mechanism to define a wide range of test objectives. HTOL comes with a formal semantics, and can encode all standard criteria but strong mutations. Besides specification, HTOL is appealing in the context of test automation as it allows to handle criteria in a unified way. As a first practical application, we present a universal coverage measurement tool supporting a wide range of standard criteria. Initial experiments demonstrate that the tool is practical and scales on realistic programs.

## I. INTRODUCTION

**Context.** In current software engineering practice, testing [1], [2], [3], [4] remains the primary approach to find bugs in a piece of code. We focus here on *white-box software testing*, in which the tester has access to the source code – as it is the case for example in unit testing. As testing all the possible program inputs is intractable in practice, the software testing community has notably defined *code-coverage criteria* (a.k.a. *adequacy criteria* or *testing criteria*) [3], [4], to select test inputs to be used. In regulated domains such as aeronautics, these coverage criteria are strict normative requirements that the tester must satisfy before delivering the software. In other domains, coverage criteria are recognized as a good practice for testing, and a key ingredient of test-driven development.

A coverage criterion fundamentally specifies a set of *test requirements* or *objectives*, which should be fulfilled by the selected test inputs. Typical requirements include for example covering all statements (statement coverage criterion) or all branches in the code (decision coverage criterion). These requirements are essential to an automated white-box testing process, as they are used to guide the selection of new test cases, decide when testing should stop and assess the quality of a *test suite* (i.e., a set of test cases including test inputs). In automated white-box testing, a *coverage measurement tool* is used to establish which proportion of the requirements are actually covered by a given test suite, while a *test generation tool* tries to generate automatically a test suite satisfying the requirements of a given criterion.

**Problem.** Dozens of code-coverage criteria have been proposed in the literature [3], [4], from basic control-flow or data-

flow [5] criteria to mutations [6] and MCDC [7], offering notably different ratios between testing thoroughness and effort. However, from a technical standpoint, these criteria are seen as very dissimilar bases for automation, so that most testing tools (coverage measurement or test generation) are restricted to a very small subset of criteria (cf. Table I) and that supporting a new criterion is time-consuming. *Hence, the wide variety and deep sophistication of coverage criteria in academic literature is barely exploited in practice, and academic criteria have only a weak penetration into industry.*

**Goal and challenges.** We intend to bridge the gap between the potentialities offered by the huge body of academic work on (code-)coverage criteria on one side, and their limited use in the industry on the other side. In particular, we aim at proposing a *well-defined and unifying specification mechanism for these criteria*, enabling a clear separation of concerns between the precise declaration of test requirements on one side, and the automation of white-box testing on the other side. This is a *fruitful* approach that has been successfully applied for example with SQL for databases and with temporal logics for model checking. This is also a *challenging* task as such a mechanism should be, at the same time: (1) well-defined, (2) expressive enough to encode test requirements from most existing criteria, and (3) amenable to automation – coverage measurement and/or test generation.

**Proposal.** We introduce *hyperlabels*, a generic specification language for white-box test requirements. Technically, hyperlabels are a major extension of *labels* proposed by Bardin et al. [8]. While labels can express a large range of criteria [8] (including a large part **WM'** of weak mutations [9], and a weak variant of **MCDC** [10]), they are still too limited in terms of expressiveness, not being able for example to express strong variants of **MCDC** [7] or most dataflow criteria [5]. In contrast, hyperlabels are able to encode *all criteria from the literature* [4] but full mutations [6], [9].

Compared with similar previous attempts, hyperlabels try to find a sweetspot between genericity, specialization to coverage criteria and automation. Indeed, FQL [11] cannot encode **MCDC** or **WM'** but provides automatic test generation [12], while temporal logics such as HyperLTL or HyperCTL\* [13] are so expressive that automation faces significant scalability issues. Hyperlabels are both *necessary* and (almost) *sufficient* for expressing all interesting coverage criteria, and they seem

to be amenable to *efficient* automation.

**Contribution.** The four main contributions of this paper are:

1. We introduce a *novel taxonomy of coverage criteria* (Section IV), orthogonal to both the standard classification [3] and the one by Ammann and Offutt [4]. Our classification is *semantical*, based on the nature of the reachability constraints underlying a given criterion. This view is sufficient for classifying all existing criteria but mutations, and yields new insights into coverage criteria, emphasizing the complexity gap between a given criterion and basic reachability. A visual representation of this taxonomy is proposed, *the cube of coverage criteria*<sup>1</sup>;

2. We propose HTOL (Hyperlabel Test Objective Language), a formal specification language for test objectives (Section V) based on *hyperlabels*. While labels reside in the cube origin, our language adds new constructs for combining (atomic) labels, *allowing us to encode any criterion from the cube taxonomy*. We present HTOL’s syntax and give a formal semantics in terms of coverage. Finally, we give a few encodings of criteria beyond labels. Notably, HTOL can express subtle differences between the variants of **MCDC** (Section V-D1);

3. As a first application of hyperlabels, and in order to demonstrate their expressiveness, we provide in Section VI a list of encodings for *almost all code coverage criteria defined in the Ammann and Offutt book [4]*, including many criteria beyond labels (cf. Table II). The only missing criteria are strong mutations and full weak mutations, yet a large subset of weak mutations can be encoded [8].

4. As a second application of hyperlabels, and in order to demonstrate their practicality, we present the design and implementation of a universal and easily extensible code coverage measurement tool (Section VII) based on HTOL. The tool already supports *in a unified way* fourteen coverage criteria, including all criteria from Table I and six which are beyond labels. We report on several experiments demonstrating that the approach is efficient enough and scales well, both in terms of program size and number of tests.

**Potential impact and future work.** Hyperlabels provide a *lingua franca* for defining, extending and comparing criteria in a clearly documented way, as well as a specification language for writing universal, extensible and interoperable testing tools. By making the whole variety and sophistication of academic coverage criteria much more easily accessible in practice, hyperlabels help bridging the gap between the rich body of academic results in criterion-based testing and their limited use in the industry.

We intend to develop a test generation tool dedicated to hyperlabels in a middle term. Actually, automatic test generation can already be obtained by combining test generation for atomic labels [8] with coverage measurement for hyperlabels, yet a more dedicated technique is certainly desirable.

TABLE I  
CRITERIA SUPPORTED IN A FEW POPULAR COVERAGE TOOLS

Tool / Criterion	FC	BBC	DC	CC	DCC	MCDC	BPC
Gcov	✓	✓	✓				
Bullseye	✓				✓		
Parasoft	✓	✓	✓	✓		✓	✓
Semantic Designs	✓		✓				
Testwell CTC++	✓	✓			✓	✓	

FC: functions, BBC: basic blocks, DC: decisions, CC: conditions, DCC: decision condition, MCDC: modified decision condition, BPC: basis paths

## II. OVERVIEW

We briefly sketch in Figure 1 the workflow of our universal coverage measurement tool, in order to give an idea of how HTOL helps to build test automation tools supporting a wide range of coverage criteria.

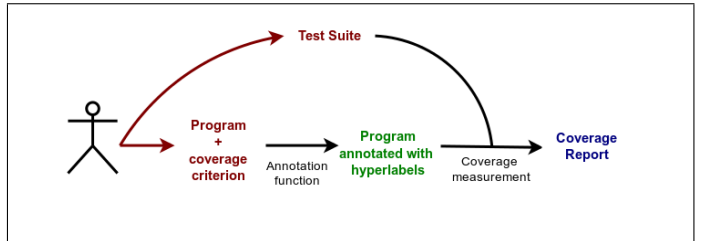


Fig. 1: Workflow of the universal coverage measurement tool

The user provides a program under test  $P$  and a test suite  $TS$ , selects a coverage criterion  $\mathbb{C}$  among a list of supported criteria and obtains the coverage score of the test suite for the criterion  $\mathbb{C}$ . *Internally*, the program  $P$  is first *automatically annotated* with hyperlabels *representing exactly the coverage objectives* defined by  $\mathbb{C}$  (cf. Figure 2 for an example) – we call *annotation function* (or *labeling function*) such a transformation, then coverage achieved by  $TS$  is *measured on the annotated program* and reported to the user.

From a developer’s point of view, the analysis engine (here, coverage measurement) is written once and for all (*shared among criteria*), and supporting a new criterion simply comes down to write a new annotation function (*shared among analysis engines*).

## III. BACKGROUND

### A. Basics: Programs, Tests and Coverage

We give here a formal definition of coverage and coverage criteria, following [8]. Given a program  $P$  over a vector  $V$  of  $m$  input variables taking values in a domain  $D \triangleq D_1 \times \dots \times D_m$ , a *test datum*  $t$  for  $P$  is a valuation of  $V$ , i.e.  $t \in D$ . A *test suite*  $TS \subseteq D$  is a finite set of test data. A (finite) execution of  $P$  over some  $t$ , denoted  $P(t)$ , is a (finite) run  $\sigma \triangleq \langle (loc_0, s_0), \dots, (loc_n, s_n) \rangle$  where the  $loc_i$  denote successive (control-)locations of  $P$  ( $\approx$  statements of the programming language in which  $P$  is written) and the  $s_i$  denote the successive internal states of  $P$  ( $\approx$  valuation of all global and local variables and of all memory-allocated

<sup>1</sup>By analogy to the  $\lambda$ -cube of functional programming.

structures) after the execution of each  $loc_i$  ( $loc_0$  refers to the initial program state).

A test datum  $t$  reaches a location  $loc$  at step  $k$  with internal state  $s$ , denoted  $t \rightsquigarrow_P^k \langle loc, s \rangle$ , if  $P(t)$  has the form  $\sigma \cdot \langle loc, s \rangle \cdot \rho$  where  $\sigma$  is a partial run of length  $k$ . When focusing on reachability, we omit  $k$  and write  $t \rightsquigarrow_P \langle loc, s \rangle$ .

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  $t \rightsquigarrow_P \mathbf{c}$ . A (source-code based) coverage criterion  $\mathbb{C}$  is defined as a systematic way of deriving a set of test objectives  $\mathbf{C} = \mathbb{C}(P)$  for any program under test  $P$ . A test suite  $TS$  satisfies (or achieves) a coverage criterion  $\mathbb{C}$  if  $TS$  covers  $\mathbb{C}(P)$ . When there is no ambiguity, we identify the coverage criterion  $\mathbb{C}$  for a given program  $P$  with the derived set of test objectives  $\mathbf{C} = \mathbb{C}(P)$ .

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

Finally, for a test suite  $TS$  and a set  $\mathbf{C}$  of test objectives, the coverage score of  $TS$  w.r.t.  $\mathbf{C}$  is the ratio of the number of test objectives in  $\mathbf{C}$  covered by  $TS$  to its cardinality  $|\mathbf{C}|$ . The coverage score of  $TS$  w.r.t. a coverage criterion  $\mathbb{C}$  is then its coverage score w.r.t. the set  $\mathbf{C} = \mathbb{C}(P)$ .

### B. A Quick Tour of Coverage Criteria

A wide variety of criteria have been proposed in the literature [2], [4], [3]. We briefly review in this section the main criteria used throughout the paper.

*Control-flow graph coverage* criteria include basic block coverage (**BBC**, equivalent to statement coverage), branch coverage (**BC**) and several path-based criteria (where each one specifies a particular set of paths to cover in the graph) such as edge-pair (**EPC**), prime path (**PPC**), basis path (**BPC**), simple/complete round trip (**SRTC/CRTC**) and complete/specified path (**CPC/SPC**) coverage.

*Call graph coverage* criteria include notably function coverage (**FC**, all the call graph nodes, i.e. each program function should be called at least once) and call coverage (**FCC**, all the graph edges, i.e. each function should be called at least once from each of its callers).

*Data-flow coverage* [5] concerns checking that each value defined in the tested program is actually used, either by one of its possible uses (**all-defs**), or by all of them (**all-uses**), or even along any of its def-use paths (**all-du-paths**).

*Logic coverage* criteria focus on exercising various truth value combinations for the logical predicates (i.e. branching conditions) of the tested program. The most basic criteria here are decision coverage (both values for each predicate, **DC** – equivalent to **BC**), (atomic) condition coverage (both

values for each literal in each predicate, **CC**) and multiple condition coverage (all literal value combinations for each predicate, **MCC**). Advanced criteria include **MCDC** [7] and its variants [14], [4] **GACC**, **CACC** (masking MCDC) and **RACC** (unique-cause MCDC), as well as their inactive clause coverage counterparts **GICC** and **RICC**. Other criteria consider the disjunctive normal form of the predicates [4, Chap. 3.6], such as implicant coverage **IC**, unique true point coverage **UTPC** and corresponding unique true point and near false point pair coverage **CUTPNFP** [15].

Finally, in *mutation coverage* [6], test requirements address the ability to detect that each of slight syntactic variants of the tested program (the *mutants*) behaves differently from the original code. In strong mutation coverage (**SM**), the divergence must be detected in the program outputs, whereas in weak mutation coverage (**WM**) [9] the divergence must be detected just after the mutation. Both **SM** and **WM** are very powerful [16], [17]. Recently, Bardin *et al.* identified side-effect-free weak mutations (**WM'**) [8], [18] as an expressive yet efficiently automatable fragment.

### C. Criterion Encoding with Labels

In previous work, we have introduced *labels* [8], a code annotation language to encode concrete test objectives, and shown that several common coverage criteria can be simulated by label coverage, i.e. given a program  $P$  and a criterion  $\mathbf{C}$ , the concrete test objectives instantiated from  $\mathbf{C}$  for  $P$  can always be encoded using labels. As our main contribution is a major extension of labels into hyperlabels, we recall here basic results about labels.

**Labels.** Given a program  $P$ , a *label*  $\ell \in \text{Labs}_P$  is a pair  $\langle loc, \varphi \rangle$  where  $loc$  is a location of  $P$  and  $\varphi$  is a predicate over the internal state at  $loc$ , that is, such that: (1)  $\varphi$  contains only variables and expressions (using in the same language as  $P$ ) defined at location  $loc$  in  $P$ , and (2)  $\varphi$  contains no side-effect expressions. There can be several labels defined at a single location, which can possibly share the same predicate. More concretely, our labels can be compared to labels in the C language, decorated with a pure C expression.

We say that a test datum  $t$  covers a label  $\ell \triangleq \langle loc, \varphi \rangle$  in  $P$ , denoted  $t \rightsquigarrow_P \ell$ , if there is a state  $s$  such that  $t$  reaches  $\langle loc, s \rangle$  (i.e.  $t \rightsquigarrow_P \langle loc, s \rangle$ ) and  $s$  satisfies  $\varphi$ . An *annotated program* is a pair  $\langle P, L \rangle$  where  $P$  is a program and  $L \subseteq \text{Labs}_P$  is a set of labels for  $P$ . Given an annotated program  $\langle P, L \rangle$ , we say that a test suite  $TS$  satisfies the *label coverage criterion* (**LC**) for  $\langle P, L \rangle$ , denoted  $TS \rightsquigarrow_{\langle P, L \rangle} \text{LC}$ , if  $TS$  covers every label of  $L$  (i.e.  $\forall \ell \in L : \exists t \in TS : t \rightsquigarrow_P \ell$ ).

**Criterion Encoding.** Label coverage *simulates a coverage criterion*  $\mathbf{C}$  if any program  $P$  can be *automatically* annotated with a set of labels  $L$  in such a way that any test suite  $TS$  satisfies **LC** for  $\langle P, L \rangle$  if and only if  $TS$  covers all the concrete test objectives instantiated from  $\mathbf{C}$  for  $P$ . We call *annotation* (or *labeling*) *function* such a procedure automatically adding test objectives into a given program for a given coverage criterion.

It is shown in [8] that label coverage can notably simulate basic-block coverage (**BBC**), branch coverage (**BC**) and

decision coverage (**DC**), function coverage (**FC**), condition coverage (**CC**), decision condition coverage (**DCC**), multiple condition coverage (**MCC**) as well as the side-effect-free fragment of weak mutations (**WM'**). The encoding of **GACC** can also be deduced from [10]. Figure 2 illustrates the simulation of some criteria with labels on sample code – that is, the resulting annotated code automatically produced by the corresponding annotation functions.

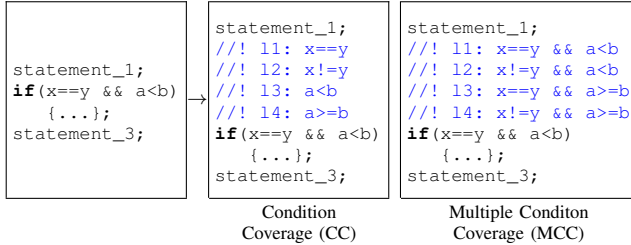


Fig. 2. Encoding of standard test requirements with labels (from [8])

The main benefit of labels is to *unify* the treatment of test requirements belonging to different classes of coverage criteria in a transparent way, thanks to the *automatic insertion* of labels in the program under test.

**Limits.** A label can only express the requirement that an assertion at a single location in the code must be covered by a single test execution. This is not expressive enough to encode the test objectives coming from path-based criteria, data-flow criteria, strong variants of **MCDC** or full mutations.

**Our goal.** In this work, we aim at extending the expressive power of labels towards all the criteria presented in Section III-B (except **WM** and **SM**). The proposed extension should preserve the automation capabilities of labels.

#### IV. A NEW TAXONOMY: THE CUBE

We propose a new taxonomy for code coverage criteria, based on the semantics of the associated reachability problem<sup>2</sup>. We take standard reachability constraints as a basis, and consider three orthogonal extensions:

- Basis** location-based reachability, constraining a single program location and a single test execution at a time,
- Ext1** reachability constraints relating several executions of the same program (hyperproperties [19]),
- Ext2** reachability constraints along a whole execution path (safety [20]),
- Ext3** reachability constraints involving choices between several objectives.

The basis corresponds to criteria that can be encoded with labels. Extensions 1, 2 and 3 can be seen as three euclidean axes that spawn from the basis and add new capabilities to labels along three orthogonal directions. This gives birth to a visual representation of our taxonomy as a cube, depicted in Figure IV, where each coverage criterion from Section III-B (but mutations) is arranged on one of the cube vertices,

<sup>2</sup>More precisely: the reachability problem of the test requirements associated to the coverage criterion.

depending on the expressiveness of its associated reachability constraints. Intuitively, strong mutation falls outside the cube because it relates two executions on *two programs*, the program under test and the mutant. Yet, we can classify test objectives corresponding to the violation of security properties such as non-interference (cf. Example 4, Section V-B).

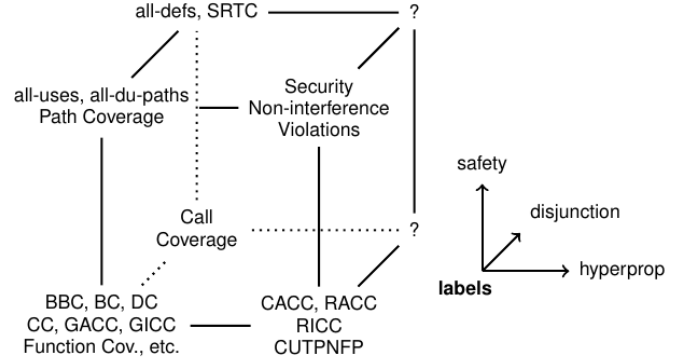


Fig. 3. The “cube” taxonomy of coverage criteria

This taxonomy is interesting in several respects. First, it is *semantic*, in the sense that it refers to the reachability problems underlying the test requirements rather than to the artifact which the test requirements are drawn from. In that sense it represents progress toward abstraction compared to the older taxonomies [4], [3], the one of [4] being already more abstract than [3]. Second, it is very concise (only three basic parameters) and yet almost comprehensive, yielding new insights on criteria, through their distance to basic reachability. Interestingly, while many criteria require two extensions, we do not know of any criterion involving the three extensions. More generally, no criterion seems to be using a disjunction of constraints over several executions of the same program.

#### V. HYPERLABELS

The previous section shows that our semantic taxonomy is suitable to represent the whole set of coverage criteria we are interested in. Since labels correspond to basic reachability constraints, we seek to extend them in the three directions of axes in order to build a universal test requirement description language. We detail here the principle, syntax and semantics of the proposed HTOL language.

##### A. Principles

HTOL is based on labels [8] (referred to as *atomic* now) to which we add five constructions, namely: *bindings*, *sequences*, *guards*, *conjunctions* and *disjunctions*. By combining these operators over atomic labels, one builds new objectives to be covered, which we call *hyperlabels*.

- Bindings  $\ell \triangleright \{v_1 \leftarrow e_1; \dots\}$  store in *meta-variable(s)*  $v_1, \dots$  the value of well-defined expression(s)  $e_1, \dots$  at the state at which atomic label  $\ell$  is covered;

- Sequence  $\ell_1 \xrightarrow{\phi} \ell_2$  requires two atomic labels  $\ell_1$  and  $\ell_2$  to be covered sequentially by a single test run, constraining the whole path section between them by  $\phi$ ;
- Conjunction  $h_1 \cdot h_2$  requires two hyperlabels  $h_1, h_2$  to be covered by (possibly distinct) test cases, enabling to express *hyperproperties* about sets of tests;
- Disjunction  $h_1 + h_2$  requires covering at least one of hyperlabels  $h_1, h_2$ . This enables to simulate criteria involving disjunctions of objectives;
- Guard  $\langle h \mid \psi \rangle$  expresses a constraint  $\psi$  over meta-variables observed (at different locations and/or during distinct executions) when covering labels underlying  $h$ .

## B. First Examples

We present here a first few examples of criterion encodings using hyperlabels. They are presented in an informal way, a formal semantics of hyperlabels being given in Section V-C.

**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 demonstrating 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:

$$\begin{aligned}
l &\triangleq (loc_1, d) \triangleright \{c_1 \leftarrow x==y; c_2 \leftarrow a<b\} \\
l' &\triangleq (loc_1, \neg d) \triangleright \{c'_1 \leftarrow x==y; c'_2 \leftarrow a<b\} \\
h_1 &\triangleq \langle l \cdot l' \mid c_1 \neq c'_1 \wedge c_2 = c'_2 \rangle \\
h_2 &\triangleq \langle l \cdot l' \mid c_1 = c'_1 \wedge c_2 \neq c'_2 \rangle
\end{aligned}$$

$h_1$  requires that the test suite reaches  $loc_1$  twice (through the  $\cdot$  operator) – with one or two tests but different values for decision  $d$ . The values taken by the atomic conditions when  $loc_1$  is reached are bound (through  $\triangleright$ ) to metavariables  $c_1, c_2$  (first execution) and  $c'_1, c'_2$  (second one). Moreover, these recorded values must satisfy the guard  $c_1 \neq c'_1 \wedge c_2 = c'_2$ , meaning that  $c_1$  alone can influence the decision. Similarly,  $h_2$  ensures the desired test objective for  $c_2$ .

**Example 2 (Call coverage)** Let us continue by showing the interest of the disjunction operator. Consider the following code snippet where  $f$  and  $g$  are two functions.

```
int f() {
if (...) { /* loc_1 */ g(); }
if (...) { /* loc_2 */ g(); }}
```

The function call coverage criterion (**FCC**) requires a test case going from  $f$  to  $g$ , i.e. passing either through  $loc_1$  or  $loc_2$ . This is exactly represented by hyperlabel  $h_3$  below:

$$h_3 \triangleq (loc_1, true) + (loc_2, true)$$

**Example 3 (all-uses)** We illustrate now the sequence operator  $\rightarrow$ . 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-uses** dataflow criterion for the definition of variable  $a$  at line  $loc_1$ , a test suite must cover the two def-use paths from  $loc_1$  to  $loc_2$  and to  $loc_3$ . These two objectives are represented by hyperlabels  $h_4 \triangleq (loc_1, true) \rightarrow (loc_2, true)$  and  $h_5 \triangleq (loc_1, true) \rightarrow (loc_3, true)$ .

**Example 4 (Non-interference)** Last, we present a more demanding example that involves bindings, sequences and guards. *Non-interference* is a strict security policy model which prescribes that information does not flow between sensitive data (*high*) towards non-sensitive data (*low*). This is a typical example of hypersafety property [19], [13]. Hyperlabels can express the violation of such a property in a straightforward manner. Consider the code snippet below.

```
int flowcontrol(int high, int low) {
// loc 1
{...}
// loc 2
return res; }
```

Non-interference is violated here if and only if two executions with the same `low` input exhibit different output (`res`) – because it would mean that a difference in the `high` input is observable. This can be encoded with hyperlabel  $h_6$ :

$$\begin{aligned}
l_1 &\triangleq (loc_1, true) \triangleright \{lo \leftarrow low\} \rightarrow (loc_2, true) \triangleright \{r \leftarrow res\} \\
l_2 &\triangleq (loc_1, true) \triangleright \{lo' \leftarrow low\} \rightarrow (loc_2, true) \triangleright \{r' \leftarrow res\} \\
h_6 &\triangleq \langle l_1 \cdot l_2 \mid lo = lo' \wedge r \neq r' \rangle
\end{aligned}$$

## C. Formal definition

**Syntax.** The syntax is given in Figure 3, where:

- $\ell \triangleq \langle loc, \varphi \rangle \in \text{Labs}_P$  is an atomic label.
- $B \in \text{Bindings}_{loc}$  is a partial mapping between arbitrary metavariable names  $v \in \text{HVars}$  and well-defined expressions  $e$  at the program location  $loc$ ;
- $l, l_1, \dots, l_i, \dots, l_n$  are atomic labels with bindings;
- $\phi_i$  is a predicate over the metavariable names defined in the bindings of labels  $l_1, \dots, l_i$ , over the current program location  $pc$  ( $\approx$  program counter) and over the variable names defined in all program locations that can be executed in a path going from  $loc_i$  to  $loc_{i+1}$ .
- $h, h_1, h_2 \in \text{Hyps}_P$  are hyperlabels;
- $\psi$  is a predicate over the set  $\text{nm}(h)$  of *h-visible names* (i.e. metavariable names *guaranteed* to be recorded by  $h$ 's bindings), defined as follows:

$$\begin{aligned}
\text{nm}(\ell \triangleright B) &\triangleq \text{all the names defined in } B \\
\text{nm}([l_1 \xrightarrow{\phi_1} \dots l_n]) &\triangleq \text{nm}(l_1) \cup \dots \cup \text{nm}(l_n) \\
\text{nm}(\langle h \mid \psi \rangle) &\triangleq \text{nm}(h) \\
\text{nm}(h_1 \cdot h_2) &\triangleq \text{nm}(h_1) \cup \text{nm}(h_2) \\
\text{nm}(h_1 + h_2) &\triangleq \text{nm}(h_1) \cap \text{nm}(h_2);
\end{aligned}$$

$h ::= l$	label
$  [l_1 \xrightarrow{\phi_1} \{l_i \xrightarrow{\phi_i}\}^* l_n]$	sequence of labels
$  \langle h \mid \psi \rangle$	guarded hyperlabel
$  h_1 \cdot h_2$	conjunction of hyperlabels
$  h_1 + h_2$	disjunction of hyperlabels
$l ::= \ell \triangleright B$	atomic label with bindings
$B ::= \{v_1 \leftarrow e_1; \dots\}$	bindings

**Fig. 3:** Syntax of Hyperlabels

**Well-formed hyperlabels.** In general, a name can be bound multiple times in a single hyperlabel, which would result in ambiguities when evaluating guards. To prevent this issue, we define in Figure 4 a well-formed predicate  $\text{wf}(\cdot)$  over hyperlabels. In the remaining part of this paper, we will only consider well-formed hyperlabels.

$\frac{\forall i, j, i \neq j \Rightarrow v_i \neq v_j}{\text{wf}(\ell \triangleright \{v_1 \leftarrow e_1; \dots; v_n \leftarrow e_n\})}$	$\frac{\text{wf}(h)}{\text{wf}(\langle h \mid \psi \rangle)}$
$\frac{\forall i, j, i \neq j \Rightarrow \text{nm}(l_i) \cap \text{nm}(l_j) = \emptyset}{\text{wf}([l_1 \xrightarrow{\phi_1} \dots l_n])}$	
$\frac{\text{wf}(h_1) \quad \text{wf}(h_2) \quad \text{nm}(l_1) \cap \text{nm}(l_2) = \emptyset}{\text{wf}(h_1 \cdot h_2)}$	
$\frac{\text{wf}(h_1) \quad \text{wf}(h_2) \quad \text{nm}(l_1) = \text{nm}(l_2)}{\text{wf}(h_1 + h_2)}$	

**Fig. 4:** Well-formed hyperlabels

In particular, on well-formed hyperlabels,  $\text{nm}$  is compatible with distributivity of  $\cdot$  and  $+$ . For instance, if we have  $\text{wf}(h)$  with  $h \triangleq h_1 \cdot (h_2 + h_3)$ , then, with  $h' \triangleq (h_1 \cdot h_2) + (h_1 \cdot h_3)$ , we have  $\text{wf}(h')$  and  $\text{nm}(h) = \text{nm}(h')$ .

**Semantics.** HTOL is given a semantics in terms of *coverage* and *execution traces*, as was done for atomic labels [8]. This kind of semantic is not tied to syntactic elements of the program under test, allowing for example to express **WM**<sup>?</sup>.

A primary requirement for covering hyperlabels is to capture execution states into the variables defined in bindings. For that, we introduce the notion of *environment*. An environment  $\mathcal{E} \in \text{Envs}$  is a partial mapping between names and values, that is,  $\text{Envs} \triangleq \text{HVars} \rightarrow \text{Values}$ . Given an execution state  $s$  at the program location  $\text{loc}$  and some bindings  $B \in \text{Bindings}_{\text{loc}}$ , the *evaluation* of  $B$  at state  $s$ , noted  $\llbracket B \rrbracket_s$  is an environment  $\mathcal{E} \in \text{Envs}$  such that  $\mathcal{E}(v) = \text{val}$  iff  $B(v)$  evaluates to  $\text{val}$  considering the execution state  $s$ .

We can now define *hyperlabel coverage*. A test suite  $TS$  covers a hyperlabel  $h \in \text{Hyps}_P$ , noted  $TS \overset{\text{H}}{\rightsquigarrow}_P h$ , if there

exists some environment  $\mathcal{E} \in \text{Envs}$  such that the pair  $\langle TS, \mathcal{E} \rangle$  covers  $h$ , noted  $\langle TS, \mathcal{E} \rangle \overset{\text{H}}{\rightsquigarrow}_P h$ , defined by the inference rules of Figure 5. An *annotated program* is a pair  $\langle P, H \rangle$  where  $P$  is a program and  $H \subseteq \text{Hyps}_P$  is a set of hyperlabels for  $P$ . Given an annotated program  $\langle P, H \rangle$ , we say that a test suite  $TS$  satisfies the *hyperlabel coverage criterion* (**HLC**) for  $\langle P, H \rangle$ , noted  $TS \overset{\text{H}}{\rightsquigarrow}_{\langle P, H \rangle} \text{HLC}$  if the test suite  $TS$  covers every hyperlabel from  $H$  (i.e.  $\forall h \in H : TS \overset{\text{H}}{\rightsquigarrow}_P h$ ).

The criterion simulation introduced for labels [8] is generalized to hyperlabels. Hyperlabel coverage simulates a coverage criterion **C** if any program  $P$  can be automatically annotated with a set of hyperlabels  $H$ , so that, for any test suite  $TS$ ,  $TS$  satisfies **HLC** for  $\langle P, H \rangle$  iff  $TS$  fulfills all the concrete test objectives instantiated from **C** for  $P$ .

**Disjunctive Normal Form.** Any well-formed hyperlabel  $h$  can be rewritten into a *disjunctive normal form* (DNF), i.e. a *coverage-equivalent* hyperlabel  $h_{dnf}$  arranged as a disjunction  $h_{dnf} \triangleq c_1 + \dots + c_i + \dots + c_n$  of *guarded conjunctions*  $c_i \triangleq \langle ls_1^i \dots ls_p^i \mid \psi(B_{ls_1^i}, \dots, B_{ls_p^i}) \rangle$  over atomic labels or sequences. The equivalence between  $h$  and  $h_{dnf}$  is stated as

$$\forall TS \subseteq D \forall \mathcal{E} \in \text{Envs}, \langle TS, \mathcal{E} \rangle \overset{\text{H}}{\rightsquigarrow}_P h \Leftrightarrow \langle TS, \mathcal{E} \rangle \overset{\text{H}}{\rightsquigarrow}_P h_{dnf}.$$

DNF normalization is an important step of our coverage measurement algorithm (cf. Section VII-A).

#### D. Advanced Examples

1) *Playing with MCDC variants:* Example 1 provides an encoding of the strongest version of **MCDC** (a.k.a. **RACC**). Yet, weaker variants exist. Encoding them into hyperlabels helps clarify the subtle differences between those variants.

**GACC** (General Active Clause Coverage) is the weakest variant of **MCDC**. It is also the sole variant encodable with atomic labels [10]. Let us assume that we have a predicate  $p$  composed of  $n$  atomic conditions  $c_1, \dots, c_n$ . **GACC** requires that for each  $c_i$ , the test suite triggers two distinct executions of the predicate: one where  $c_i$  is true, one where  $c_i$  is false, and both such that the truth value of  $c_i$  impacts the truth value of the whole predicate. Yet, it is not required that switching the value of  $c_i$  is indeed feasible, and the two executions do not have to be correlated. Going back to the code snippet of Example 1, **GACC** requirement for  $c_1$  can be simulated by  $l_3$  and  $l_4$ , where  $d(x, y)$  denotes decision  $d$  (cf. Example 1) where  $c_1$  and  $c_2$  are replaced by  $x$  and  $y$ .

$$l_3 \triangleq (\text{loc}_1, c_1 \wedge d(\text{true}, c_2) \neq d(\text{false}, c_2))$$

$$l_4 \triangleq (\text{loc}_1, \neg c_1 \wedge d(\text{true}, c_2) \neq d(\text{false}, c_2))$$

**CACC** (Correlated Active Clause Coverage), or masking **MCDC** is stronger than **GACC**. It includes every requirement from **GACC** and additionally requires that for each clause  $c_i$ , the two executions are such that if  $p$  is true (resp. false) in the first one, then it is false (resp. true) in the second one. **CACC** cannot be encoded into atomic labels because of this last requirement that correlates the two executions together. Yet, it can be encoded with hyperlabels. Using the same code as in Example 1, **CACC** requirement for  $c_1$  can be simulated

LABEL		GUARD		CONJUNCTION	
$t \in TS$	$t \rightsquigarrow_P^k \langle loc, s \rangle$	$s \models \varphi$	$\mathcal{E} \supseteq \llbracket B \rrbracket_s$	$\langle TS, \mathcal{E} \rangle \rightsquigarrow_P h$	$\mathcal{E} \models \psi$
$t \rightsquigarrow_{\mathcal{E}}^k \langle loc, \varphi \rangle \triangleright B$		$\langle TS, \mathcal{E} \rangle \rightsquigarrow_P \langle loc, \varphi \rangle \triangleright B$		$\langle TS, \mathcal{E} \rangle \rightsquigarrow_P \langle h \mid \psi \rangle$	
$\langle TS, \mathcal{E} \rangle \rightsquigarrow_P h_1 \cdot h_2$		$\langle TS, \mathcal{E} \rangle \rightsquigarrow_P h_1 \cdot h_2$		$\langle TS, \mathcal{E} \rangle \rightsquigarrow_P h_1 \cdot h_2$	
DISJUNCTION LEFT		DISJUNCTION RIGHT		SEQUENCE	
$\langle TS, \mathcal{E} \rangle \rightsquigarrow_P h_1$	$\langle TS, \mathcal{E} \rangle \rightsquigarrow_P h_2$	$t \in TS$	$\forall i \in [1, n], t \rightsquigarrow_{\mathcal{E}}^{k_i} l_i$	$\forall i \in [1, n-1], k_i < k_{i+1}$	$\forall i \in [1, n-1], \forall j \in ]k_i, k_{i+1}[, (loc_j, s_j) = P(t)_j \wedge \phi_i(\mathcal{E}, loc_j, s_j)$
$\langle TS, \mathcal{E} \rangle \rightsquigarrow_P h_1 + h_2$		$\langle TS, \mathcal{E} \rangle \rightsquigarrow_P h_1 + h_2$		$\langle TS, \mathcal{E} \rangle \rightsquigarrow_P [l_1 \xrightarrow{\phi_1} \{l_i \xrightarrow{\phi_i}\}^* l_n]$	

Naming convention:  $TS$  test suite;  $\mathcal{E}$  hyperlabel environment;  $h, h_1, h_2$  hyperlabels;  $\psi$  hyperlabel guard predicate;  $n$  positive integer;  $l_1, \dots, l_n$  atomic labels with bindings;  $t$  test datum;  $k, k_1, \dots, k_n$  execution step numbers;  $loc_j, loc$  program locations;  $s_j, s$  execution states;  $P(t)_j$  the  $j$ -th step of the program run  $P(t)$  of  $P$  on  $t$ ;  $\phi_1, \dots, \phi_n$  predicates over sequences of labels;  $\varphi$  label predicate;  $B$  hyperlabel bindings.

Fig. 5. Inference rules for hyperlabel semantics

by the following hyperlabel  $h_7$ , built on the two atomic labels  $l_3$  and  $l_4$  defined for **GACC**:

$$h_7 \triangleq \langle l_3 \triangleright \{r \leftarrow d\} \cdot l_4 \triangleright \{r' \leftarrow d\} \mid r \neq r' \rangle$$

2) *More DataFlow criteria*: The **all-defs** coverage criterion requires that each definition of a variable must be connected to *one of its* uses. The criterion adds a disjunction of objectives to the **all-uses** criterion. Going back to Example 3, the **all-defs** requirement for the definition of variable  $a$  at line  $loc_1$  can be simply simulated by hyperlabel  $h_8 \triangleq h_4 + h_5$ .

Finally, data-flow criteria can be refined to consider the **definition and use of single array cells**, while the standard approach considers arrays as a whole. Indeed, the index of the accessed cells may not be known statically, making it impossible to relate defs and uses, as well as to define def-free paths without dynamic information. For example, in the following code, the path from  $loc_1$  to  $loc_3$  is a valid du-path iff  $i = k \neq j$ , which cannot be known statically:

```
int foo(int i, int j, int k) {
  /* loc_1 */ a[i] = x;
  /* loc_2 */ a[j] = y;
  /* loc_3 */ z = a[k] + 1; }
```

With hyperlabels, we just have to add bindings to the atomic labels for saving the values of  $i$  and  $j$  and use the guard operator to force them being equal. Encoding for the previous example is given below, with  $pc$  the current line of code:

$$l_5 \triangleq (loc_1, true) \quad l_6 \triangleq (loc_3, true)$$

$$h_9 \triangleq \langle l_5 \triangleright \{v_1 \leftarrow i\} \xrightarrow{pc=loc_2} \Rightarrow_{j \neq v_1} l_6 \triangleright \{v_2 \leftarrow k\} \mid v_1 = v_2 \rangle$$

3) *Path-based Criteria*: Most test objectives coming from path-based criteria have a straightforward encoding with the  $\rightarrow$  operator, typically **complete path coverage** (for a finite number of paths). A few criteria also require operator  $+$  for choices between paths, e.g. **simple round trip coverage**.

## VI. EXTENSIVE CRITERIA ENCODING

As a first application of hyperlabels, we perform an extensive literature review and we try to encode all coverage criteria with hyperlabels. Especially, we have been able to encode all criteria from the Ammann and Offutt book [4],

but strong mutations and full weak mutations. Indeed, these two criteria really require the ability to run tests on variants of the original program, whereas HTOL does not modify the code itself. These results are summarized in Table II, where we also specify which criteria can be expressed by atomic labels alone, and the required hyperlabel operators otherwise.

TABLE II  
SIMULATION OF CRITERIA FROM [4].

	Encodable by				See Sec. or ref.
	labels	hyperlabels using			
	$\xrightarrow{\phi}$	$\mid \psi$	$\cdot$	$+$	
<b>Control-flow graph coverage</b> Statement, Basic-Block, Branch <i>Path coverage</i> : EPC, PPC, CRTC, CPC, SPC Simple Round Trip coverage	✓				[8] V-D3 V-D3
<b>Call-graph coverage</b> Function coverage (all nodes) Call coverage (all edges)	✓				III-C 2
<b>Data-flow coverage</b> All Definitions (all-defs) + array cell definitions		•			V-D2 V-D2
All Uses (all-uses) + array cell definitions		•	•		3 V-D2
All Def-Use Paths (all-du-paths) + array cell definitions		•	•		V-D3 V-D2
<b>Logic expression coverage</b> BBC, CC, DCC, MCC	✓				[8]
<i>MCDC variants</i> : GACC, GICC CACC, RACC, RICC	✓		•	•	V-D1, [10] V-D1
<i>DNF-based criteria</i> : IC, UTPC CUTPNFPPC	✓		•	•	
<b>Mutation coverage</b> Side-effect-free Weak Mut. (Full) Weak Mut., Strong Mut.	✓				[8]
		not encodable			

✓: expressible by atomic labels      •: required hyperlabel operators

Interestingly, many criteria fall beyond the scope of atomic labels, and many also require combining two or three HTOL operators. This is a strong *a posteriori* evidence that the language of hyperlabel is both *necessary* and (almost) *sufficient* to encode state-of-the-art coverage criteria. *Detailed encodings are available on the companion website*<sup>3</sup>.

<sup>3</sup>Companion website: <http://icst17.marcozzi.net>

## VII. UNIVERSAL COVERAGE MEASUREMENT TOOL

As a second application, we describe a *universal* coverage measurement tool, built on HTOL, following the view of Section II and the philosophy of LTest [21]. The two basic building blocks are: (1) a coverage measurement procedure for test suites on programs annotated with hyperlabels, and (2) pre-defined (hyperlabeling) annotation functions for standard criteria (cf. Section VI).

This prototype is the first coverage measurement tool able to handle *all* coverage criteria from [4] (but strongest mutation variants) in a *unified way*. Fourteen criteria are supported so far – their annotation functions are provided: six based on hyperlabels (**CACC**, **RACC**, **FCC**, **BPC**, **all-defs** and **all-use**), plus eight based on atomic labels<sup>4</sup>. Supporting a new coverage criterion amounts to implement its annotation function.

While our coverage measurement algorithm runs in worst-case exponential time considering the whole HTOL expressiveness, experiments demonstrate that the tool is efficient enough on existing coverage criteria, and scales well with both program size and number of tests.

### A. Computing the coverage of a test suite

Given an annotated program  $\langle P, H \rangle$  and a test suite  $TS$ , our coverage measurement algorithm follows three steps.

**normalization** First, each hyperlabel  $h \in H$  is rewritten into its *disjunctive normal form* (cf. Section V-C).

**harvesting** Second, each test case  $t$  from  $TS$  is run on  $P$ . Every atomic label and label sequence covered during the run is saved on-the-fly, together with the environment (values of metavariables) that instantiates the label’s bindings at the coverage points.

**consolidation** Third, the collected coverage information is propagated within the syntax tree (in DNF) of every  $h \in H$ , in order to establish if  $TS$  covers  $h$  or not.

These steps are now described in more details.

**Normalization.** As stated in Section V-C, any (well-formed) hyperlabel  $h$  can be rewritten into an equivalent hyperlabel  $h_{dnf}$  in disjunctive normal form. This form of labels is both very convenient for coverage measurement and very common in practice. This is done by applying the rewrite rules of Figure 6 bottom-up from the leaves of the hyperlabel tree. The proof of equivalence between  $h$  and  $h_{dnf}$  can easily be obtained by induction on  $h$ .

**Environment harvesting.** Once hyperlabels in DNF have been obtained, each test  $t$  from the suite  $TS$  is run on  $P$ , and the coverage information for basic labels, sequences and binding values is collected. Note that we need to store all possible binding values encountered along the execution of  $t$ , 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. Due to space limitations, we do not describe this point here, common in runtime monitoring *A detailed description is available on the companion website*<sup>5</sup>.

<sup>4</sup>Namely: **FC**, **BBC**, **DC**, **CC**, **DCC**, **MCC**, **GACC** and **WM**.

<sup>5</sup>Companion website: <http://icst17.marcozzi.net>

$$\begin{array}{l}
 \overline{l \rightsquigarrow \langle l | true \rangle} \qquad \overline{s \rightsquigarrow \langle s | true \rangle} \qquad \overline{h \rightsquigarrow \sum_i \langle \pi_i | \psi_i \rangle} \\
 \langle h | \psi \rangle \rightsquigarrow \sum_i \langle \pi_i | \psi_i \wedge \psi \rangle \\
 \\
 h^L \rightsquigarrow \sum_i \langle \pi_i^L | \psi_i^L \rangle \qquad h^R \rightsquigarrow \sum_j \langle \pi_j^R | \psi_j^R \rangle \\
 \overline{h^L + h^R \rightsquigarrow \sum_i \langle \pi_i^L | \psi_i^L \rangle + \sum_j \langle \pi_j^R | \psi_j^R \rangle} \\
 \\
 h^L \rightsquigarrow \sum_i \langle \pi_i^L | \psi_i^L \rangle \qquad h^R \rightsquigarrow \sum_j \langle \pi_j^R | \psi_j^R \rangle \\
 \overline{h^L \cdot h^R \rightsquigarrow \sum_i \sum_j \langle \pi_i^L \cdot \pi_j^R | \psi_i^L \wedge \psi_j^R \rangle} \qquad \text{notation: } \pi \triangleq l \dots s
 \end{array}$$

Fig. 6: Rewriting hyperlabel into DNF

**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  $c \triangleq \langle ls_1 \dots ls_p \mid \psi(B_{ls_1}, \dots, B_{ls_p}) \rangle$  is covered iff each label or sequence  $ls_j, j \in 1..p$  is saved as covered in  $E$  and there is at least one set of environments  $\mathcal{E}_j \in E$  (one for every  $ls_j$  with bindings) such that  $\psi(\mathcal{E}_1, \dots, \mathcal{E}_p)$  is true;
- a disjunction  $h_{hnf} \triangleq c_1 + \dots + c_i + \dots + c_n$  is covered iff at least one of the  $c_i$  is covered.

In practice, the tool tries every possible combination of  $\mathcal{E}_j$  from  $E$  for every  $c_i$ , until it finds one which makes  $\psi$  true (in which case  $TS$  covers  $h$ ) or proves that none exists (in which case  $h$  is not covered by  $TS$ ). Note that under the assumption that  $P$  terminates for all test cases in  $TS$ , this algorithm is both correct and complete with respect to HTOL’s semantics, that is, it will find a combination covering a well-formed hyperlabel  $h$  iff  $TS \rightsquigarrow_P h$ . This can be shown by induction on the number of disjunctions and conjunctions in the normalized form of  $h$ .

**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 algorithm 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, our algorithm appears to *perform well on existing classes of testing requirements* (cf. Section VII-C).



Here are a few explanations. First, criteria as encoded in the previous sections are naturally in DNF. Second, the critical parameters indicated above are strongly limited 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.

### B. Implementation

We have implemented a basic hyperlabel support in LTest [21], an open-source all-in-one testing platform for C programs, developed as a Frama-C [22] plugin. LTest is built on standard labels, and provides (labeling) annotation functions to automatically encode requirements from common coverage criteria, coverage measurement, automatic coverage-oriented test generation [8] and automatic detection of infeasible requirements [18]. LTest relies on PathCrawler [23] for test generation and on Frama-C for static analysis.

Our prototype extends LTest in two aspects. First, we provide an *hyperlabeling mechanism*, together with (hyperlabeling) annotation functions dedicated to the supported criteria. Second, we have implemented coverage measurement for hyperlabels.

### C. Experimentations

**Objective.** We want to assess the practical applicability of our universal coverage measurement tool, at least for unit testing.

**[RQ 1]** Is the proposed unified approach practical and efficient enough? More precisely, how does the tool scale with large test suites on criteria beyond labels?

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

- 5 small witness functions, mainly from Siemens [24], Verisec [25] & MediaBench [26], as already used in [8];
- 5 functions from OpenSSL 1.0.2 [27], a 250kloc open-source application. We focus on modules of about 1kloc.
- 3 functions from SQLite 3.13 [28], a 215kloc open-source application. We focus on modules of a few kloc.

The C files automatically annotated with HTOL test objectives are available on companion website <http://icst17.marcozzi.net>.

For **[RQ 1]**, 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, which can also be downloaded from the companion website. Each tool run is repeated 7 times. First, tests are executed without measurement (baseline), and then measuring coverage for the **CC** and **GACC** label-encodable criteria (witness). Second, tests are measured 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.

**Results and discussion.** Our main results are presented in Figure 7. Detailed results are part of the annexes and *full results are available on the companion website*.

**[RQ 1]** Figure 7 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 our implementation defines. However, many of these objectives are trivial or redundant, which could be detected using some control-flow analysis in an optimized version of the tool.

**Conclusion.** These results indicate that upgrading labels with hyperlabels makes it possible to build an (almost) universal coverage measurement tool, without losing practical applicability. The measurement time for criteria beyond labels is acceptable and remains linear with the size of the test suite. Moreover, as our tool implementation is not optimized, there is still room for a strong reduction of coverage measurement time, when using the approach in a more industrial context.

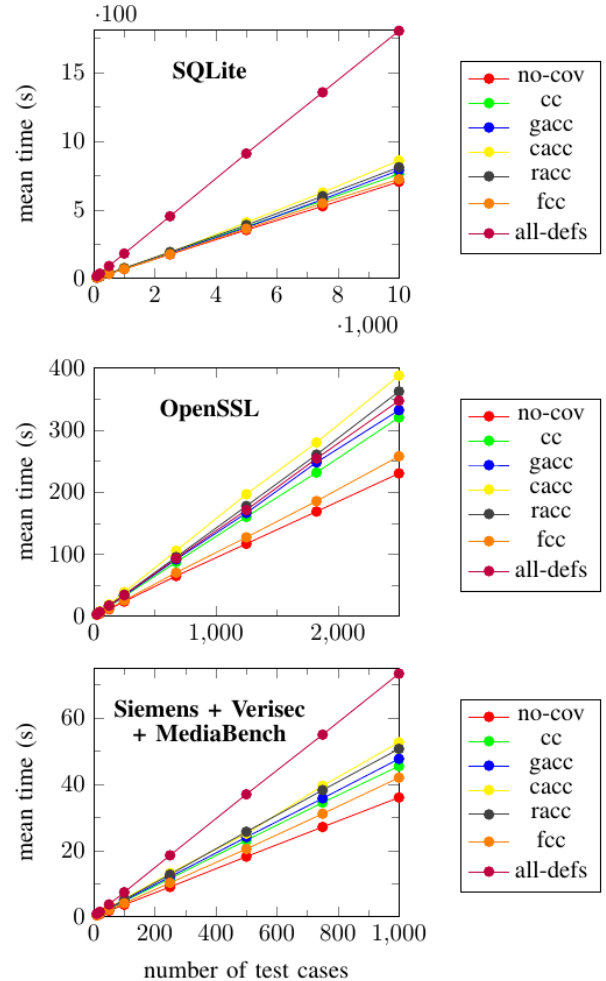


Fig. 7. Scalability of Coverage Measurement

## VIII. RELATED WORK

The two closest works to ours are *labels* [8] and *FQL*. Since the difference with labels has already been presented (Sections III-C and V-A, Table II), we focus here on FQL.

**Specification of white-box coverage criteria.** The *Fshell Query Language* (FQL) by Holzer *et al.* [11] for test suite specification and the associated Fshell [12] tool represent the closest work to ours. FQL 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. Fshell takes advantage of an off-the-shelf model-checker to automatically generate from a C program a test suite satisfying a given FQL specification.

The scope of criteria that can be encoded in FQL is incomparable with the one offered by HTOL, as FQL handles complex safety-based test requirements but no hyperproperty-based requirement. Moreover, FQL is limited to syntactic elements of the program under analysis. As a consequence, FQL cannot encode neither **MCDC** nor **WM'**.

Yet, FQL offers the interesting ability to encode, in an elegant and standardized way, generic coverage criteria (independently of any concrete program), where HTOL encodes concrete test objectives (i.e. particular instantiations of coverage criteria for given programs). Note also that FShell provides automatic test generation, while we focuses on coverage measurement for now.

**Specification of model-based coverage criteria.** Blom *et al.* [29] proposes to specify test objectives on extended finite state machines (EFSMs) as observer automata with parameters, while Hong *et al.* [30] considers CTL temporal logic. Formal encodings have also been proposed for several model-based coverage criteria in different other formalisms, like set theory [31], graph theory [32], predicate logic [33], [34], OCL [35] and Z [36]. However, for each formalism, the scope of supported criteria is limited to safety-based criteria, with no support of hyperproperties.

**Coverage objectives and hyperproperties.** Hyperproperties [19] are properties over several traces of a system. Testing hyperproperties is a rising issue, notably in the frame of security [37]. However, research in the topic still remains exploratory. Rayadurgam *et al.* [38] suggests that **MCDC** can be encoded with temporal logics, by writing the formulas for a self-composition of the tested model with itself. The paper reports that model-checking the obtained formulas rapidly faces scalability issues. Clarkson *et al.* [13] introduces HyperLTL and HyperCTL\*, which are extensions of temporal logics for hyperproperties, as well as an associated model-checking algorithm. This work makes no reference to test criterion encoding, but the proposed logics could be used to provide [30] with the ability to encode criteria like **MCDC**. However, the complexity results and first experiments [13] indicate that the approach faces strong scalability limits. HTOL being *a priori* less generic (yet, sufficient in practice), it is likely to be more amenable to *efficient* automation. In future work, we

intend to explore how HTOL formally compares to HyperLTL and HyperCTL\*.

**Test description languages.** Some languages have been designed to support the implementation of test harnesses at the program (TSTL [39], UDITA [40]) or model (TTCN-3 [41], UML Testing Profile [42]) level. A test harness is the helper code that executes the testing process in practice, which notably includes test definition, documentation, execution and logging. These languages offer general primitives to write and execute easily test suites, but independently of any explicit reference to a coverage criterion.

**Coverage measurement tools.** Code coverage is used extensively in the industry. As a result, there exists a lot of testing tools that embed some sort of coverage measurement. For instance, in 2007, a survey [43] found ten tools for programs written in the C language: Bullseye [44], CodeTEST, Dynamic [45], eXVantage, Gcov (part of GCC) [46], Intel Code Coverage Tool [47], Parasoft [48], Rational PurifyPlus, Semantic Designs [49], TCAT [50]. To this date, there are even more tools, such as COVTOOL, LDRACover [51], and Testwell CTC++ [52].

As a rule of thumb, these tools support a limited number of test criteria in a hard-coded, non-generic manner. Table I (Section I) summarizes implemented criteria for some popular tools. Our prototype already supports all these criteria in a generic and extensible way, plus seven other criteria (cf. Section VII-B). However, to be fair, 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.

## IX. CONCLUSIONS

To sum up, HTOL proposes a unified framework for describing and comparing most existing test coverage criteria. This enables in particular implementing generic tools that can be used for a wide range of criteria. We propose as a first application a universal coverage measurement tool, with an overhead sufficiently low to not be a concern in practice. Future work includes the efficient lifting of automatic test generation technologies to HTOL.

## REFERENCES

- [1] G. J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing*, 3rd ed. Wiley, 2011.
- [2] A. P. Mathur, *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.*, vol. 29, no. 4, 1997.
- [4] P. Ammann and J. Offutt, *Introduction to Software Testing*, 1st ed. Cambridge University Press, 2008.
- [5] J. W. Laski and B. Korel, "A data flow oriented program testing strategy," *IEEE Trans. Software Eng.*, vol. 9, no. 3, 1983.
- [6] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, 1978.
- [7] J. J. Chilenski and S. P. Miller, "Applicability of modified condition/decision coverage to software testing," *Software Engineering Journal*, 1994.
- [8] S. Bardin, N. Kosmatov, and F. Cheynier, "Efficient leveraging of symbolic execution to advanced coverage criteria," in *ICST*, 2014.
- [9] W. E. Howden, "Weak mutation testing and completeness of test sets," *IEEE Trans. Software Eng.*, vol. 8, no. 4, 1982.

- [10] R. Pandita, T. Xie, N. Tillmann, and J. de Halleux, "Guided test generation for coverage criteria," in *ICSM*, 2010.
- [11] A. Holzer, C. Schallhart, M. Tautschnig, and H. Veith, "How did you specify your test suite," in *ASE*, 2010.
- [12] —, "Fshell: Systematic test case generation for dynamic analysis and measurement," in *CAV*, 2008.
- [13] M. R. Clarkson, B. Finkbeiner, M. Koleini, K. K. Micinski, M. N. Rabe, and C. Sánchez, "Temporal logics for hyperproperties," in *POST*, 2014.
- [14] P. Ammann, A. J. Offutt, and H. Huang, "Coverage criteria for logical expressions," in *14th International Symposium on Software Reliability Engineering (ISSRE 2003)*. IEEE Computer Society, 2003.
- [15] T. Y. Chen and M. F. Lau, "Test case selection strategies based on boolean specifications," *Softw. Test., Verif. Reliab.*, vol. 11, no. 3, 2001.
- [16] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *ICSE*, 2005.
- [17] A. J. Offutt and S. D. Lee, "An empirical evaluation of weak mutation," *IEEE Trans. Software Eng.*, vol. 20, no. 5, 1994.
- [18] S. Bardin, M. Delahaye, R. David, N. Kosmatov, M. Papadakis, Y. L. Traon, and J. Marion, "Sound and quasi-complete detection of infeasible test requirements," in *ICST*, 2015.
- [19] M. R. Clarkson and F. B. Schneider, "Hyperproperties," *J. Comput. Secur.*, vol. 18, no. 6, 2010.
- [20] Z. Manna, *The Temporal Logic of Reactive and Concurrent Systems Specification*. Springer, 1992.
- [21] S. Bardin, O. Chebaro, M. Delahaye, and N. Kosmatov, "An all-in-one toolkit for automated white-box testing," in *TAP*. Springer, 2014.
- [22] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-C: A Program Analysis Perspective," *Formal Aspects of Computing Journal*, 2015.
- [23] 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 CS, 2004.
- [24] H. Do, S. Elbaum, and G. Rothermel, "Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and Its Potential Impact," *Empirical Software Engineering*, vol. 10, no. 4, Oct. 2005.
- [25] K. Ku, T. E. Hart, M. Chechik, and D. Lie, "A Buffer Overflow Benchmark for Software Model Checkers," in *ASE*. ACM, 2007.
- [26] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: a tool for evaluating and synthesizing multimedia and communications systems," in *ACM International Symposium on Microarchitecture, 1997*, Dec. 1997.
- [27] "OpenSSL," <https://www.openssl.org>.
- [28] "SQLite," <https://www.sqlite.org>.
- [29] J. Blom, A. Hessel, B. Jonsson, and P. Pettersson, "Specifying and generating test cases using observer automata," in *FATES*, 2005.
- [30] H. S. Hong, I. Lee, O. Sokolsky, and H. Ural, "A temporal logic based theory of test coverage and generation," in *TACAS*, 2002.
- [31] P. G. Frankl and E. J. Weyuker, "A formal analysis of the fault-detecting ability of testing methods," *IEEE Trans. Softw. Eng.*, vol. 19, no. 3, 1993.
- [32] A. Podgurski and L. A. Clarke, "A formal model of program dependences and its implications for software testing, debugging, and maintenance," *IEEE Trans. Softw. Eng.*, vol. 16, no. 9, 1990.
- [33] K.-C. Tai, "Theory of fault-based predicate testing for computer programs," *IEEE Trans. Softw. Eng.*, vol. 22, no. 8, 1996.
- [34] A. Abdurazik, P. Amman, W. Ding, and J. Offutt, "Evaluation of three specification-based testing criteria," in *ICECCS*, 2000.
- [35] M. Friske, B.-H. Schlingloff, and S. Weißleder, "Composition of model-based test coverage criteria," in *MBEES*, 2008.
- [36] S. A. Vilkomir and J. P. Bowen, "From MC/DC to RC/DC: Formalization and analysis of control-flow testing criteria," in *Formal Methods and Testing: An Outcome of the FORTEST Network, Revised Selected Papers*. Springer, 2008.
- [37] J. Kinder, "Hypertesting: The case for automated testing of hyperproperties," in *3rd Workshop on Hot Issues in Security Principles and Trust (HotSpot)*, 2015.
- [38] S. Rayadurgam and M. P. Heimdahl, "Generating MC/DC adequate test sequences through model checking," in *SEW*, 2003.
- [39] A. Groce, J. Pinto, P. Azimi, and P. Mittal, "Tstl: A language and tool for testing (demo)," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: ACM, 2015. [Online]. Available: <http://doi.acm.org/10.1145/2771783.2784769>
- [40] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov, "Test generation through programming in udita," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010. [Online]. Available: <http://doi.acm.org/10.1145/1806799.1806835>
- [41] J. Grabowski, D. Hogrefe, G. Réthy, I. Schieferdecker, A. Wiles, and C. Willcock, "An introduction to the testing and test control notation (ttn-3)," *Comput. Netw.*, vol. 42, no. 3, Jun. 2003. [Online]. Available: [http://dx.doi.org/10.1016/S1389-1286\(03\)00249-4](http://dx.doi.org/10.1016/S1389-1286(03)00249-4)
- [42] I. Schieferdecker, Z. R. Dai, J. Grabowski, and A. Rennoch, "The uml 2.0 testing profile and its relation to ttn-3," in *Proceedings of the 15th IFIP International Conference on Testing of Communicating Systems*, ser. TestCom'03. Berlin, Heidelberg: Springer-Verlag, 2003. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1764575.1764585>
- [43] Q. Yang, J. J. Li, and D. M. Weiss, "A survey of coverage-based testing tools," *The Computer Journal*, vol. 52, no. 5, 2009.
- [44] "Bullseye Testing Technology: BullseyeCoverage," <http://bullseye.com/>.
- [45] "Dynamic Code Coverage," <http://dynamic-memory.com/>.
- [46] "GCC's Gcov," <https://gcc.gnu.org/onlinedocs/gcc/Gcov.html>.
- [47] "Intel Code Coverage Tool in Intel C++ compiler," <https://software.intel.com/en-us/node/512810>.
- [48] "Parasoft C/C++test: Comprehensive dev. testing tool for C/C++," <https://www.parasoft.com/product/cpptest/>.
- [49] "Semantic designs: C test coverage tool," <http://semanticdesigns.com/Products/TestCoverage/CTestCoverage.htm>.
- [50] "Testworks: TCAT C/C++," <http://www.testworks.com/Products/Coverage/tcat.html>.
- [51] "LDRA – LDRACover," <http://www.ldra.com/en/ldrcover>.
- [52] "Testwell CTC++: Test coverage analyzer for C/C++," <http://www.testwell.fi/ctcdesc.html>.