HAL
archives-ouvertes.fr

Generic and Effective Specification of Structural Test Objectives

Sébastien Bardin, Mickaël Delahaye, Nikolai Kosmatov, Michaël Marcozzi, Virgile Prévosto

▶ To cite this version:

Sébastien Bardin, Mickaël Delahaye, Nikolai Kosmatov, Michaël Marcozzi, Virgile Prévosto. Generic and Effective Specification of Structural Test Objectives. 2016. cea-01357487

HAL Id: cea-01357487
https://hal-cea.archives-ouvertes.fr/cea-01357487

Preprint submitted on 2 Sep 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Copyright

Generic and Effective Specification of Structural Test Objectives

Sébastien Bardin Mickaël Delahaye Nikolai Kosmatov
Michaël Marcozzi Virgile Prevosto
CEA, LIST, Software Reliability Laboratory, PC 174
91191 Gif-sur-Yvette France
firstname.lastname@cea.fr

ABSTRACT

While a wide range of different, sometimes heterogeneous test coverage criteria have been proposed, there exists no generic formalism to describe them, and available test automation tools usually support only a small subset of them. We introduce a unified specification language, called HTOL, providing a powerful generic mechanism to define test objectives, which permits encoding numerous existing criteria and supporting them in a unified way. HTOL comes with a formal semantics and can express complex requirements over several executions (using a novel notion of *hyperlabels*), as well as alternative requirements or requirements over a whole program execution. A novel classification of a large class of existing criteria is proposed. Finally, a coverage measurement tool for HTOL objectives has been implemented. Initial experiments suggest that the proposed approach is both efficient and practical.

CCS Concepts

•**Software and its engineering** → **Dynamic analysis; Software testing and debugging;** *Software reliability;* Domain specific languages;

Keywords

white-box software testing, code coverage, test requirements, specification language

1. INTRODUCTION

Context. In current software engineering practice, testing [36, 35, 50, 3] 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*) [50, 3], 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 [50, 3], from basic control-flow or data-flow [31] criteria to mutations [13] and MCDC [10], 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 1) and that supporting a new criterion is time-consuming. *As a consequence, the wide variety and deep sophistication of code-coverage criteria in the academic literature is barely exploited in practice, and academic criteria have a weak penetration into the industrial world.*

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) very expressive (to encode test requirements from most existing criteria), and (3) amenable to automation (coverage measurement and 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.* [6]. While labels can express a large range of criteria [6] (including a large part of weak mutations [26] and a weak variant of **MCDC** [38]), they are still too limited in terms of expressiveness, not being able for example to express strong variants of **MCDC** [10] or most dataflow criteria [31]. In contrast, hyperlabels are able to encode *all criteria from the literature* [3] but full mutations [13, 26].

We also propose a universal coverage measurement tool supporting hyperlabels, and demonstrate that the label-based test generation algorithm from [6] yields acceptable (and automatic) coverage results for hyperlabels.

Contribution. The four main contributions of this paper are the following:
1. We introduce a *novel taxonomy of coverage criteria* (Section 3), orthogonal to both the standard classification [50] and the one by Ammann and Offutt [3]. Our classification is *semantical*, based on the nature of the reachability constraints underlying a given coverage 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*[1];
2. We propose HTOL, a formal specification language for test objectives (Section 4) based on *hyperlabels*. While labels reside in the cube origin, our language adds new constructs in order to combine (atomic) labels, *allowing us to encode any criterion from the cube taxonomy*. We present the syntax of the language 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 4.4.1);
3. As a first application of hyperlabels, and in order to demonstrate their expressiveness, we provide in Section 5 a list of encodings for *almost all code coverage criteria defined in the Ammann and Offutt book [3]*, including many criteria beyond labels (cf. Table 2). The only missing criteria are strong mutations and weak mutations, yet a large subset of weak mutations can be encoded [6].
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 6) based on HTOL. The tool already supports *in a unified way* fourteen coverage criteria, including all criteria from Table 1 and six which are beyond labels. We report on several experiments demonstrating that the approach is efficient enough and combines well with label-based automatic test generation.

Potential impact. 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 variety 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.

2. BACKGROUND

2.1 Basics: Programs, Tests and Coverage

We give here a formal definition of coverage and coverage criteria, following [6].

Given a program P over a vector V of m input variables taking values in a domain $D \triangleq D_1 \times \cdots \times D_m$, a *test datum* t for P is a valuation of V, i.e. $t \in D$. 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 $\sigma \triangleq \langle (loc_0, s_0), \ldots, (loc_n, s_n) \rangle$ where the loc_i denote successive (control-)locations of P ($\approx$ line of code) and the s_i denote the successive internal states of P ($\approx$ valuation of all global and local variables and of all memory-allocated structures) after the execution of each loc_i (loc_0 refers to the initial program state).

[1]By analogy to the λ-cube of functional programming.

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

Table 1: Criteria supported in some coverage tools

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

Given a test objective $\mathbf{c}$, we write $t \leadsto_P \mathbf{c}$ if test datum t covers $\mathbf{c}$. We extend the notation for a test suite TS and a set of test objectives $\mathbf{C}$, writing $TS \leadsto_P \mathbf{C}$ when for any $\mathbf{c} \in \mathbf{C}$, there exists $t \in TS$ such that $t \leadsto_P \mathbf{c}$. A *(source-code based) coverage criterion* $\mathbb{C}$ is defined as a systematic way of deriving a set of test objectives $\mathbf{C} = \mathbb{C}(P)$ for any program under test P. A test suite TS satisfies (or achieves) a 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 $\mathbf{c} \triangleq (loc, \texttt{cond})$ or $\mathbf{c} \triangleq (loc, \texttt{!cond})$, where `cond` is the condition of the branching statement at location loc, and $t \leadsto_P \mathbf{c}$ if t reaches some (loc, S) such that `cond` evaluates to *true* (resp. *false*) in S.

Finally, 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 $\mathbf{C} = \mathbb{C}(P)$.

2.2 A Quick Tour of Coverage Criteria

A wide variety of criteria have been proposed in the literature [35, 3, 50]. 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 [31] 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** [10] and its variants [2, 3] **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 [3, 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** [9].

Finally, in *mutation coverage* [13], 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 (**SMC**), the divergence must be detected in the program outputs, whereas in weak mutation coverage (**WMC**) [26] the divergence must be detected just after the mutation. Both **SMC** and **WMC** are very powerful coverage criteria [4, 37].

2.3 Criterion Encoding with Labels

In previous work, Bardin *et al.* have introduced *labels* [6], 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* $\ell \in \mathsf{Labs}$ is a pair $\langle loc, \varphi \rangle$ where loc is a location of P and φ is a predicate over the internal state at loc, that is, such that: (1) φ contains only variables and expressions defined at location loc in P, and (2) φ contains no side-effect expressions. There can be several labels defined at a single location, which can possibly share the same predicate.

We say that a test datum t *covers a label* $\ell \triangleq \langle loc, \varphi \rangle$ in P, denoted $t \overset{\mathrm{L}}{\leadsto}_P \ell$, if there is a state s such that t reaches $\langle loc, s \rangle$ (i.e. $t \leadsto_P \langle loc, s \rangle$) and s satisfies φ. An *annotated program* is a pair $\langle P, L \rangle$ where P is a program and $L \subseteq \mathsf{Labs}$ 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 $TS \overset{\mathrm{L}}{\leadsto}_{\langle P,L \rangle}$ **LC**, if TS covers every label of L (i.e. $\forall \ell \in L : \exists t \in TS : t \overset{\mathrm{L}}{\leadsto}_P \ell$).

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.

It is shown in [6] 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. The encoding of **GACC** can also be deduced from [38]. Figure 1 illustrates the simulation of some criteria with labels on sample code.

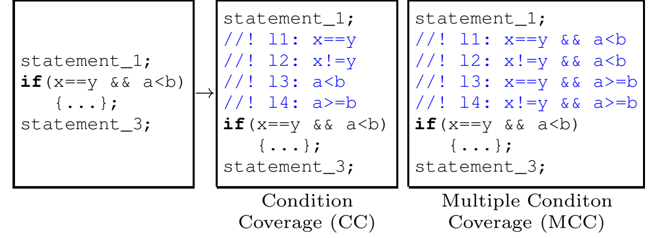

Figure 1: Encoding of standard test requirements with labels (from [6])

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 2.2 (except **WMC** and **SMC**). The proposed extension should preserve the automation capabilities of labels.

3. A NEW TAXONOMY: THE CUBE

We propose a new taxonomy for code coverage criteria, based on the semantics of the associated reachability problem[2]. 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 [12]),

Ext2 reachability constraints along a whole execution path (safety [34]),

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 2, where each coverage criterion from Section 2.2 (but mutations) is arranged on one of the cube vertices, depending on the expressiveness of its associated reachability constraints. We can also classify test objectives corresponding to the the violation of security properties such as non-interference (cf. Example 4, Section 4.2).

[2]More precisely: the reachability problem of the test requirements associated to the coverage criterion.

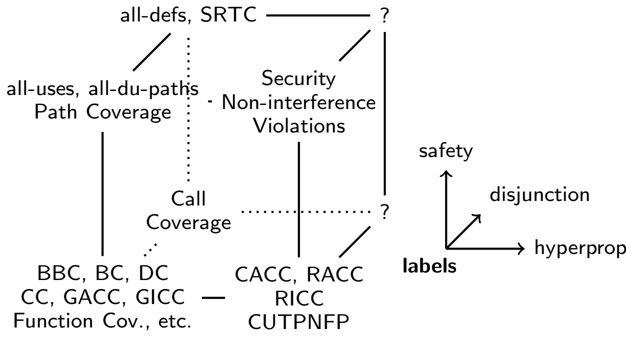

Figure 2: The "cube" taxonomy of coverage criteria

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 a progress toward abstraction compared to the older taxonomies [3, 50], the one of [3] being already more abstract than [50]. 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, many standard criteria require two extensions, yet we do not have any example of criterion involving the three extensions.

4. 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 language, called HTOL (Hyperlabel Test Objective Language).

4.1 Principles

HTOL is based on labels [6] (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 $\ell \triangleright \{v_1 \hookleftarrow e_1; \ldots\}$ store in *meta-variable(s)* $v_1, \ldots$ the value of well-defined expression(s) $e_1, \ldots$ at the state at which atomic label ℓ is covered;
- Sequence $\ell_1 \xrightarrow{\phi} \ell_2$ requires two atomic labels ℓ_1 and ℓ_2 to be covered sequentially by a single test run, constraining the whole path section between them by ϕ;
- 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 to cover 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 ψ over meta-variables observed (at different locations and/or during distinct executions) when covering labels underlying h.

4.2 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 4.3.

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:

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

h_1 requires that the test suite reaches loc_1 twice (through the $\cdot$ operator), 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 (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 $\xrightarrow{\cdot}$. 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 [12, 11]. Hyperlabels can express the violation of such a property in a straighforward 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:

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

4.3 Formal definition

We now formally define HTOL's syntax and semantics.

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

- $\ell \triangleq \langle loc, \varphi \rangle \in \mathsf{Labs}$ is an atomic label.
- $B \in \mathsf{Bindings}_{loc}$ is a partial mapping between arbitrary metavariable names $v \in \mathsf{HVars}$ and well-defined expressions e at the program location loc;
- $l, l_1, \cdots, l_i, \cdots, l_n$ are atomic labels with bindings;
- ϕ_i is a predicate over the metavariable names defined in the bindings of labels $l_1, \ldots, l_i$, 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}.
- $h, h_1, h_2 \in \mathsf{Hyps}$ are hyperlabels;
- ψ is a predicate over the set $\mathrm{nm}(h)$ of *h-visible names* (i.e. metavariable names *guaranteed* to be recorded by h's bindings), defined as follows:

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

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 a well-formed predicate $\mathrm{wf}(\cdot)$ over hyperlabels, whose definition is given in Figure 4.

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

In the remaining part of this paper, we will only consider well-formed hyperlabels.

h ::=	l	label
|	$[l_1 \xrightarrow{\phi_1} \{l_i \xrightarrow{\phi_i}\}^\star\ l_n]$	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 ::=	$\{v_1 \hookleftarrow e_1; \ldots\}$	bindings

Figure 3: Syntax of Hyperlabels

$$\frac{\textit{no redundant names in } B}{\mathrm{wf}(\ell \triangleright B)} \qquad \frac{\mathrm{wf}(h)}{\mathrm{wf}(\langle h \mid \psi \rangle)}$$

$$\frac{\forall i, j,\ i \neq j \Rightarrow \mathrm{nm}(l_i) \cap \mathrm{nm}(l_j) = \emptyset}{\mathrm{wf}([l_1 \xrightarrow{\phi_1} \cdots\ l_n])}$$

$$\frac{\mathrm{wf}(h_1) \qquad \mathrm{wf}(h_2) \qquad \mathrm{nm}(l_1) \cap \mathrm{nm}(l_2) = \emptyset}{\mathrm{wf}(h_1 \cdot h_2)}$$

$$\frac{\mathrm{wf}(h_1) \qquad \mathrm{wf}(h_2) \qquad \mathrm{nm}(l_1) = \mathrm{nm}(l_2)}{\mathrm{wf}(h_1 + h_2)}$$

Figure 4: Well-formed hyperlabels

Semantics. HTOL is given a semantics in terms of *coverage*. This is an extension of the semantics of atomic labels of Bardin et al. [6].

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 $\mathcal{E} \in \mathsf{Envs}$ is a partial mapping between names and values, that is, $\mathsf{Envs} \triangleq \mathsf{HVars} \nrightarrow \mathsf{Values}$. Given an execution state s at the program location loc and some bindings $B \in \mathsf{Bindings}_{loc}$, the *evaluation* of B at state s, noted $[\![B]\!]_s$ is an environment $\mathcal{E} \in \mathsf{Envs}$ such that $\mathcal{E}(v) = val$ iff $B(v)$ evaluates to val considering the execution state s.

We can now define *hyperlabel coverage*. A test suite TS *covers* a hyperlabel $h \in \mathsf{Hyps}$, noted $TS \overset{\mathrm{H}}{\leadsto}_P h$, if there exists some environment $\mathcal{E} \in \mathsf{Envs}$ such that the pair $\langle TS, \mathcal{E} \rangle$ covers h, noted $\langle TS, \mathcal{E} \rangle \overset{\mathrm{H}}{\leadsto}_P h$, defined by the inference rules of Figure 5. An *annotated program* is a pair $\langle P, H \rangle$ where P is a program and $H \subseteq \mathsf{Hyps}$ 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 $TS \overset{\mathrm{H}}{\leadsto}_{\langle P,H \rangle}$ **HLC** if the test suite TS covers every hyperlabel from H (i.e. $\forall h \in H : TS \overset{\mathrm{H}}{\leadsto}_P h$).

The notion of criterion simulation introduced for labels [6] can then be 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 $\langle P, H \rangle$ if TS fulfils all the concrete test objectives instantiated from **C** for P and conversely.

Disjunctive Normal Form. Any well-formed hyperlabel

Label
$$\frac{t \in TS \qquad t \rightsquigarrow^{k}_{P} \langle loc, s\rangle \qquad s \vDash \varphi \qquad \mathcal{E} \supseteq [\![B]\!]_s}{t \rightsquigarrow^{k}_{\mathcal{E}} \langle loc, \varphi\rangle \triangleright B \qquad \langle TS, \mathcal{E}\rangle \overset{\mathrm{H}}{\rightsquigarrow}_P \langle loc, \varphi\rangle \triangleright B}$$

Guard
$$\frac{\langle TS, \mathcal{E}\rangle \overset{\mathrm{H}}{\rightsquigarrow}_P h \qquad \mathcal{E} \vDash \psi}{\langle TS, \mathcal{E}\rangle \overset{\mathrm{H}}{\rightsquigarrow}_P \langle h \mid \psi\rangle}$$

Conjunction
$$\frac{\langle TS, \mathcal{E}\rangle \overset{\mathrm{H}}{\rightsquigarrow}_P h_1 \qquad \langle TS, \mathcal{E}\rangle \overset{\mathrm{H}}{\rightsquigarrow}_P h_2}{\langle TS, \mathcal{E}\rangle \overset{\mathrm{H}}{\rightsquigarrow}_P h_1 \cdot h_2}$$

Disjunction Left
$$\frac{\langle TS, \mathcal{E}\rangle \overset{\mathrm{H}}{\rightsquigarrow}_P h_1}{\langle TS, \mathcal{E}\rangle \overset{\mathrm{H}}{\rightsquigarrow}_P h_1 + h_2}$$

Disjunction Right
$$\frac{\langle TS, \mathcal{E}\rangle \overset{\mathrm{H}}{\rightsquigarrow}_P h_2}{\langle TS, \mathcal{E}\rangle \overset{\mathrm{H}}{\rightsquigarrow}_P h_1 + h_2}$$

Sequence
$$\frac{\begin{array}{c} t \in TS \qquad \forall i \in [1, n]\,,\ t \rightsquigarrow^{k_i}_{\mathcal{E}} l_i \qquad \forall i \in [1, n-1]\,,\ k_i < k_{i+1} \\ \forall i \in [1, n-1]\,,\ \forall j \in]k_i, k_{i+1}[\,,\ (loc_j, s_j) = P(t)_j \wedge \phi_i(\mathcal{E}, loc_j, s_j) \end{array}}{\langle TS, \mathcal{E}\rangle \overset{\mathrm{H}}{\rightsquigarrow}_P [l_1 \xrightarrow{\phi_1} \{l_i \xrightarrow{\phi_i} \}^{\star}\ l_n]}$$

Naming convention: TS test suite; $\mathcal{E}$ hyperlabel environment; h, h_1, h_2 hyperlabels; ψ 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; φ label predicate; B hyperlabel bindings.

Figure 5: Inference rules for hyperlabel semantics

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* $c_i \triangleq \langle ls^i_1 \cdot \ldots \cdot ls^i_p \mid \psi(B_{ls^i_1}, \cdots, B_{ls^i_p})\rangle$ over atomic labels or sequences. The equivalence between h and h_{dnf} is stated as

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

We provide a DNF normalization algorithm in Section 6.1 (Algorithm 6). DNF will prove convenient for defining our coverage measurement algorithm (Section 6.1).

4.4 Advanced Examples

We illustrate now a few more advanced encodings.

4.4.1 Playing with MCDC variants

We have given in Example 1 an encoding of the strongest version of **MCDC** (a.k.a. **RACC**). Yet, weaker variants exist. Encoding them into hyperlabels helps clarifying the subtle differences between those variants.

GACC (General Active Clause Coverage) is the weakest variant of **MCDC**. It is also the sole variant to be encodable with atomic labels [38]. 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 the two atomic labels l_3 and l_4

$l_3 \triangleq (loc_1, c_1 \wedge ((true \wedge c_2) \neq (false \wedge c_2)))$
$l_4 \triangleq (loc_1, \neg c_1 \wedge ((true \wedge c_2) \neq (false \wedge c_2)))$

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

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

4.4.2 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 $h_8 \triangleq h_4 + h_5$, where h_4 and h_5 are the hyperlabels defined in Example 3.

Finally, we show that 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. This is not a trivial refinement as the index of the cell manipulated at one location 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;
}
```

Such a test objective can however be perfectly encoded with hyperlabels, by adding bindings to the atomic labels for saving the values of i and j and using the guard operator to force them being equal. The encoding for the previous code is given below, where pc represents the current line of code:

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

$$h_9 \triangleq \langle l_5 \triangleright \{v_1 \hookleftarrow i\} \xrightarrow[\Rightarrow j \neq v_1]{pc=loc_2} l_6 \triangleright \{v_2 \hookleftarrow k\} \mid v_1 = v_2\rangle$$

4.4.3 Path-based Criteria

Most test objectives coming from path-based criteria can be encoded in a straightforward way with the $\rightarrow$ operator. Typically, **complete path coverage** can be encoded by considering that S is the set of all paths in P. As soon as there is a loop in P, this will produce an infinite set of hyperlabels. This is coherent with the fact that complete coverage is not finitely applicable and thus not feasible in general. A few path-based criteria require also the $+$ operator for expressing choices between several paths, such as **simple round trip coverage**.

5. 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 [3], but strong mutations and (full) weak mutations. These results are summarized in Table 2, where we also specify which criteria can be expressed by atomic labels alone, and the required hyperlabel operators otherwise.

	Encodable by					See Sec. or ref.
	labels	hyperlabels using				
		$\xrightarrow{\phi}$	$\lvert\psi\rangle$	$\cdot$	$+$	
Control-flow graph coverage						
Statement, Basic-Block, Branch	✓					[6]
Path coverage:						
EPC, PPC, CRTC, CPC, SPC		•				4.4.3
Simple Round Trip coverage		•			•	4.4.3
Call-graph coverage						
Function coverage (all nodes)	✓					2.3
Call coverage (all edges)					•	2
Data-flow coverage						
All Definitions (all-defs)		•			•	4.4.2
+ array cell definitions		•	•		•	4.4.2
All Uses (all-uses)		•				3
+ array cell definitions		•	•			4.4.2
All Def-Use Paths (all-du-paths)		•				4.4.3
+ array cell definitions		•	•			4.4.2
Logic expression coverage						
BBC, CC, DCC, MCC	✓					[6]
MCDC variants:						
GACC, GICC	✓					4.4.1, [38]
CACC, RACC, RICC			•	•		4.4.1
DNF-based criteria:						
IC, UTPC	✓					
CUTPNFPPC			•	•		
Mutation coverage						
Side-effect-free Weak Mut.	✓					[6]
(Full) Weak Mut., Strong Mut.	not encodable					

✓: expressible by atomic labels •: required hyperlabel operators

Table 2: Simulation of criteria from [3]

Interestingly, many criteria fall beyond the scope of atomic labels, and many also require to combine 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.

All detailed encodings will be made publicly available in a technical report once the paper is accepted for publication.

6. COVERAGE MEASUREMENT TOOL

As a second application, we describe a *universal* coverage measurement tool, built on HTOL. This tool is to hyperlabels what LTest [5] is to atomic labels. Namely, we first design a coverage measurement procedure for test suites on programs annotated with hyperlabels (instead of atomic labels), then universality is achieved through providing hyperlabeling encodings of standard criteria (cf. Section 5).

This prototype is the first coverage measurement tool able to handle *all* coverage criteria from [3] (but strongest mutation variants) in a *unified way.* Fourteen criteria are supported so far. While our coverage measurement algorithm runs in worst-case exponential time, experiments demonstrate that the tool is efficient enough on existing coverage criteria beyond simple labels. Finally, we show that current automatic test generation tools can already be lifted (in a weak but still useful sense) from labels to hyperlabels, by combining them with our coverage measurement tool.

6.1 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 4.3).

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 instantiate 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 4.3, 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.

$$\frac{}{l \rightsquigarrow \langle l|true\rangle} \qquad \frac{}{s \rightsquigarrow \langle s|true\rangle} \qquad \frac{h \rightsquigarrow \sum_i \langle \pi_i|\psi_i\rangle}{\langle h|\psi\rangle \rightsquigarrow \sum_i \langle \pi_i|\psi_i \wedge \psi\rangle}$$

$$\frac{h^L \rightsquigarrow \sum_i \langle \pi_i^L|\psi_i^L\rangle \qquad h^R \rightsquigarrow \sum_j \langle \pi_j^R|\psi_j^R\rangle}{h^L + h^R \rightsquigarrow \sum_i \langle \pi_i^L|\psi_i^L\rangle + \sum_j \langle \pi_j^R|\psi_j^R\rangle}$$

$$\frac{h^L \rightsquigarrow \sum_i \langle \pi_i^L|\psi_i^L\rangle \qquad h^R \rightsquigarrow \sum_j \langle \pi_j^R|\psi_j^R\rangle}{h^L \cdot h^R \rightsquigarrow \sum_i \sum_j \langle \pi_i^L \cdot \pi_j^R|\psi_i^L \wedge \psi_j^R\rangle}$$

$$\textit{notation:} \quad \pi \triangleq l \cdot \ldots \cdot s \cdot$$

Figure 6: Rewriting hyperlabel into DNF

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 basic labels, the case of 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, since it is a common issue in runtime monitoring[3].

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;

[3]A detailed description will be available in a separate technical report once the paper is accepted for publication.

- a guarded conjunction $c \triangleq \langle ls_1 \cdot ... \cdot ls_p \mid \psi(B_{ls_1}, ...B_{ls_p}) \rangle$ 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 $\psi(\mathcal{E}_1, \cdots, \mathcal{E}_p)$ is true;
- a disjunction $h_{hnf} \triangleq c_1 + \cdots + c_i + \cdots + c_n$ 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 ψ true (in which case TS covers h) or proves that none exists (in which case h is not covered by TS).

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 6.3). Here are a few explanations. First, criteria as encoded in the previous sections are naturally in DNF, hence we do not have any normalization cost. Second, the critical parameters indicated above have very strong limitations on hyperlabels coming from existing criteria: conjunctions are of length 2; sequences are either of length 2, or they do not use bindings –in that case, environment harvesting becomes a simple coverage check, linear with the length of the sequence; the domain of metavariables is often small (boolean).

6.2 Implementation

We have implemented a basic hyperlabel support in the open-source testing tool LTest [5], an all-in-one testing platform for C programs annotated with labels, developed as a Frama-C [29] plugin. LTest is built around standard labels, and provides labeling functions to automatically encode the requirements from common coverage criteria, as well as coverage measurement, automatic coverage-oriented test generation and automatic detection of infeasible test requirements. The tool is written in OCaml, and relies on PathCrawler [48] for test generation and on Frama-C for static analysis.

We have extended the *labeling mechanism* into an *hyperlabeling mechanism*, and we have lifted the coverage measurement part from labels to hyperlabels. Moreover, we have implemented hyperlabeling functions for criteria **CACC**, **RACC**, **all-def**, **all-use**, **FCC** and **BPC** in addition to the criteria already available in LTest[4]; support for other criteria is in progress. Finally, we show in Section 6.3 how LTest test generation can be combined with our new hyperlabel-based coverage measurement algorithm.

[4]Namely: **FC**, **BBC**, **DC**, **CC**, **DCC**, **MCC**, **GACC**, and a subset of **WM**.

Our prototype will be available in open-source when the paper is accepted for publication.

6.3 Experimentations

Objectives. We assess the practical applicability of our tool and evaluate its integration with label-based test generation from LTest. The addressed research questions are:

[***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?

[***RQ 2***] Does combining our tool with label-directed test generation (LTest) yield an effective full-featured test tool for criteria beyond labels?

Protocol. We consider five standard benchmark C programs from the LTest paper [6], mainly taken from the Siemens test suite [14] (`tcas`), the Verisec benchmark [30] (`get_tag` and `full_bad` from Apache source code) and MediaBench [33] (`gd` from libgd).

For [***RQ 1***], a set of 1000 test cases is randomly generated for each program. Our tool is successively run with 10, 50, 100, 250, 500, 750 and all of these test cases. Each tool run is repeated six times. First, tests are executed without measurement (baseline), and then measuring coverage for the **CC** and **GACC** label-encodable criteria (witness). Second, tests are run and measured for the **CACC**, **RACC** and **all-defs** criteria, which involve the five operators from hyperlabels.

For [***RQ 2***], LTest test generation is used on each program to produce test suites tailored for the label-encodable **CC**, **MCC** and **GACC** criteria. Our tool then measures the coverage achieved by these test suites and by random test generation (1000 test cases per program, serving as witness), for criteria **CACC** and **RACC**.

All the experiments are performed under Ubuntu Linux 14.04 on an Intel Core i7-4712HQ CPU at 2.30GHz × 8, with 16GB of RAM.

Results and discussion. Part of our results are presented in Table 3, Figure 7 and Figure 8. *Detailed test data and results will be available in open-source when the paper is accepted for publication.*

[***RQ 1***] Table 3 details, for each program and encoded criterion, the number of defined hyperlabels and labels as well as the coverage measurement time and level for 10 and 1000 test cases. 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. In all cases, the measurement time grows linearly with the number of test cases, but the slope is sharper for the criteria involving hyperlabel operators. The time overhead, w.r.t. **GACC**, is very small for **CACC** and **RACC**, which are encoded with labels close to those of **GACC**, more some bindings, guards and conjunctions. **all-defs** is encoded using sequences and disjunctions, but its tangible time overhead, compared to the other criteria, is due to the higher number of test objectives that this criterion defines.

Conclusion. These results indicate that upgrading labels with hyperlabels makes it possible to build an (almost) universal coverage measurement tool, without blowing practical applicability off. 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 opti-

		cc	gacc	cacc	racc	all-defs
trityp	#hlab	34	34	17	17	15
50 loc	#lab	34	34	34	34	178
10 t.c.	time	0.49s	0.49s	0.54s	0.52s	0.64
	cover	6/34	2/34	0/17	0/17	4/15
1000 t.c.	time	36.54s	39.05s	43.14s	41.1s	54.16s
	cover	16/34	15/34	5/17	5/17	7/15
tcas	#hlab	58	58	29	29	47
124 loc	#lab	58	58	58	58	170
10 t.c.	time	0.64s	0.63s	0.64s	0.61s	0.71s
	cover	33/58	32/58	10/29	10/29	38/47
1000 t.c.	time	47.21s	48.56s	52.22s	49.45s	58.86s
	cover	51/58	46/58	19/29	19/29	41/47
get_tag	#hlab	48	48	24	24	34
240 loc	#lab	48	48	48	48	306
10 t.c.	time	0.68s	0.71s	0.69s	0.67s	1.21s
	cover	25/48	25/48	9/24	9/24	12/34
1000 t.c.	time	49.75s	52.22s	56.31s	55.58s	97.67s
	cover	28/48	28/48	12/24	12/24	12/34
full_bad	#hlab	32	32	16	16	24
219 loc	#lab	32	32	32	32	158
10 t.c.	time	0.56s	0.57s	0.62s	0.62s	0.83s
	cover	25/32	25/32	9/16	9/16	19/24
1000 t.c.	time	42.11s	44.47s	50.67s	48.41s	62.18s
	cover	29/32	29/32	13/16	13/16	19/24
gd	#hlab	76	76	38	38	64
319 loc	#lab	76	76	76	76	728
10 t.c.	time	0.65s	0.68s	0.74s	0.72s	1.61s
	cover	21/76	19/76	7/38	7/38	14/64
1000 t.c.	time	51.77s	54.02s	60.9s	58.84s	141.48s
	cover	49/76	46/76	19/38	17/38	33/64

Table 3: Scalability of Cov. Measurement (Details)

mized, there is still room for a strong reduction of coverage measurement time, when using the approach in a more industrial context.

[***RQ 2***] Figure 8 shows the total percentage, for all the programs, of **CACC** and **RACC** objectives that are covered by the generated test suites. Label-directed test generation provides a significantly better coverage than random generation. Coverage is also better with **MCC** and **GACC** tests than with **CC** tests, the two former criteria being much stronger than the latter. The minimal and maximal **RACC**-coverage obtained on a program with **GACC**-directed test generation are 61% and 94%, with a mean of 72% (random: minimum 29%, maximum 81%, mean 53%). Note also that these figures may take into account infeasible test objectives, artificially lowering the achieved coverage results.

Conclusion. Combining our tool with LTest test generation provides a full-featured test tool, in the sense that it handles both coverage measurement and (label-based) test generation, for almost all the criteria from the literature. The previous results indicate that this full-featured tool can provide an above-random and often good coverage ratio for beyond-label criteria.

7. RELATED WORK

The two closest works to ours are *labels* [6] and *FQL*. Since the difference with labels has already been presented (Sections 2.3 and 4.1, Table 2), we focus here on FQL.

Specification of white-box coverage criteria. The *Fshell Query Language* (FQL) by Holtzer *et al.* [24] for test suite specification and the associated Fshell [23] 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 ad-

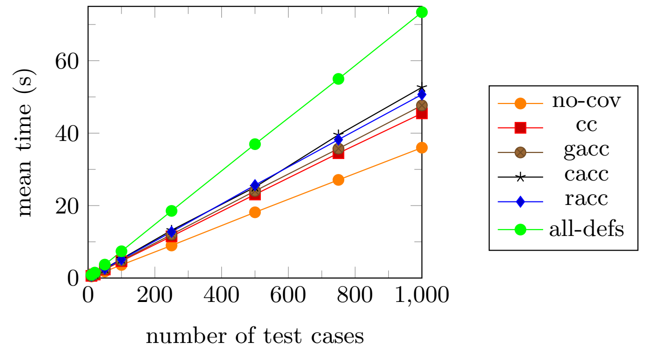

Figure 7: Scalability of Coverage Measurement

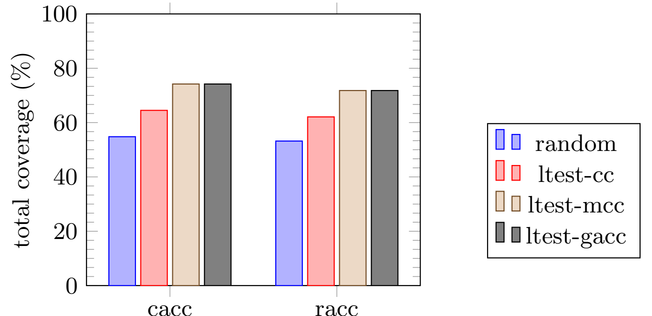

Figure 8: Coverage Efficiency of Test Generation

vantage of an off-the-shell model-checker to 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. As an important example, FQL cannot encode **MCDC**. FQL also offers the interesting ability to encode, in an elegant and standardized way, generic coverage criteria (independently of any concrete program), where HTOL is designed to encode concrete test objectives (i.e. particular instantiations of coverage criteria for a concrete program).

In future work, a promising research direction would be to use HTOL and FQL in a complementary way, by enabling the instantiation of the generic (extended) FQL specification of a criterion into a set of hyperlabels for the program to test.

Note also that FShell focus on test generation, where the tool proposed in this paper focus on coverage measurement. We intend to develop a test generation tool dedicated to hyperlabels in a middle term.

Specification of model-based coverage criteria. The work by Blom *et al.* [7] proposes to specify test objectives on extended finite state machines (EFSMs) as observer automata with parameters. Test case generation can then be expressed as a reachability problem, which can be solved with the Upaal Cover state-space exploration tool [22]. Hong *et al.* [25] also consider EFSM testing, but they encode coverage criteria as sets of formulas in the CTL temporal logic. Both papers provide encoding examples for some classical control-flow and data-flow criteria (on EFSM models). Al-

though they are theoretically more expressive than HTOL for safety-related test requirements, HTOL already offers a sufficient subset of features to encode common coverage criteria from the literature. On the other hand, these languages do not support requirements involving multiple related executions (hyperproperties), which do appear in industrial context (**MCDC**, non-interference).

Formal encodings have been proposed for several coverage criteria in different other formalisms, like set theory [16], graph theory [40], predicate logic [44, 1], OCL [17] and Z [47]. However, for each formalism, the scope of supported criteria is narrow and often limited to simple criteria.

Coverage objectives and hyperproperties. Test requirements from the strongest **MCDC** variants can be seen as examples of hyperproperties, i.e. software properties over several different traces of the system to verify. Testing hyperproperties is a rising issue, notably in the frame of security [28]. However, research in the topic still remains exploratory. Rayadurgam *et al.* [41] 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.* [11] 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 [25] with the ability to encode criteria like **MCDC**. However, the complexity results and first experiments [11] indicate that the automation of such an approach still faces strong scalability limits.

In future work, we intend to explore how HTOL formally compares to HyperLTL and HyperCTL*. As an intuition, we think that these languages are theoretically more expressive than HTOL. But while HTOL remains sufficient for encoding common code coverage criteria (including **MCDC**), it is also a strong starting point for a more general but still lightweight support of hyperproperty testing, like shown in Example 4 with non-interference.

Test description languages. Some languages have been designed to support the implementation of test harnesses at the program (TSTL [21], UDITA [19]) or model (TTCN-3 [20], UML Testing Profile [42]) level. A test harness is the helper code that will execute 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 [49] found ten tools for programs written in the C language: Bullseye [8], CodeTEST, Dynamic [15], eXVantage, Gcov (part of GCC) [18], Intel Code Coverage Tool [27], Parasoft [39], Rational PurifyPlus, Semantic Designs [43], TCAT [45]. To this date (April 2016), there are even more tools, such as COVTOOL, LDRAcover [32], and Testwell CTC++ [46]. Most existing tools only support basic coverage criteria such as statement and branch coverage. For instance, only a few like Testwell CTC++, Parasoft, and LDRAcover support **MCDC**.

As a rule of thumb, current coverage measurement tools support a limited number of test criteria in a hard-coded, non-generic manner. Table 1 (Section 1) 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 6.2).

Moreover, the lack of formalization in the reportedly supported coverage criteria prevents users to know with certainty the actually supported criteria. For example, the way shortcut logical operators are supported, or which actual flavor of **MCDC** is supported, are often left unspecified in the tools' documentation.

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.

8. 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. Furthermore, as shown in our first experiments on coverage measurements, the overhead of such tools is sufficiently low to not be a concern in practice. Future work includes the efficient lifting of automatic test generation technologies to hyperlabels, the identification of uncoverable hyperlabels, the extension of HTOL to handle mutation-based criteria, as well as the use (and extension) of HTOL for general hyperproperty testing.

9. REFERENCES

[1] A. Abdurazik, P. Amman, W. Ding, and J. Offutt. Evaluation of three specification-based testing criteria. In *Engineering of Complex Computer Systems, 2000. ICECCS 2000. Proceedings. Sixth IEEE International Conference on*, pages 179–187. IEEE, 2000.

[2] 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.

[3] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 1 edition, 2008.

[4] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, pages 402–411. ACM, 2005.

[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] S. Bardin, N. Kosmatov, and F. Cheynier. Efficient leveraging of symbolic execution to advanced coverage criteria. In *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA*, pages 173–182. IEEE Computer Society, 2014.

[7] J. Blom, A. Hessel, B. Jonsson, and P. Pettersson. Specifying and generating test cases using observer automata. In *Proceedings of the 4th International Conference on Formal Approaches to Software Testing*, FATES'04, pages 125–139, Berlin, Heidelberg, 2005. Springer-Verlag.

[8] Bullseye Testing Technology: BullseyeCoverage. http://bullseye.com/.

[9] T. Y. Chen and M. F. Lau. Test case selection strategies based on boolean specifications. *Softw. Test., Verif. Reliab.*, 11(3), 2001.

[10] J. J. Chilenski and S. P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193–200, 1994.

[11] M. R. Clarkson, B. Finkbeiner, M. Koleini, K. K. Micinski, M. N. Rabe, and C. Sánchez. *Principles of Security and Trust: Third International Conference, POST 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, chapter Temporal Logics for Hyperproperties, pages 265–284. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[12] M. R. Clarkson and F. B. Schneider. Hyperproperties. *J. Comput. Secur.*, 18(6):1157–1210, Sept. 2010.

[13] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, 1978.

[14] H. Do, S. Elbaum, and G. Rothermel. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and Its Potential Impact. *Empirical Software Engineering*, 10(4), Oct. 2005.

[15] Dynamic Code Coverage. http://dynamic-memory.com/.

[16] P. G. Frankl and E. J. Weyuker. A formal analysis of the fault-detecting ability of testing methods. *IEEE Trans. Softw. Eng.*, 19(3):202–213, Mar. 1993.

[17] M. Friske, B.-H. Schlingloff, and S. Weißleder. Composition of model-based test coverage criteria. In *MBEES*, pages 87–94, 2008.

[18] GCC's Gcov. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[19] 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*, ICSE '10, pages 225–234, New York, NY, USA, 2010. ACM.

[20] J. Grabowski, D. Hogrefe, G. Réthy, I. Schieferdecker, A. Wiles, and C. Willcock. An introduction to the testing and test control notation (ttcn-3). *Comput. Netw.*, 42(3):375–403, June 2003.

[21] 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*, ISSTA 2015, pages 414–417, New York, NY, USA, 2015. ACM.

[22] A. Hessel, K. G. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, and A. Skou. Formal methods and testing. chapter Testing Real-time Systems Using UPPAAL, pages 77–117. Springer-Verlag, Berlin, Heidelberg, 2008.

[23] A. Holzer, C. Schallhart, M. Tautschnig, and H. Veith. *Computer Aided Verification: 20th International Conference, CAV 2008 Princeton, NJ, USA, July 7-14, 2008 Proceedings*, chapter FShell: Systematic Test Case Generation for Dynamic Analysis and Measurement, pages 209–213. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[24] A. Holzer, C. Schallhart, M. Tautschnig, and H. Veith. How did you specify your test suite. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 407–416, New York, NY, USA, 2010. ACM.

[25] H. S. Hong, I. Lee, O. Sokolsky, and H. Ural. *Tools and Algorithms for the Construction and Analysis of Systems: 8th International Conference, TACAS 2002 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8–12, 2002 Proceedings*, chapter A Temporal Logic Based Theory of Test Coverage and Generation, pages 327–341. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[26] W. E. Howden. Weak mutation testing and completeness of test sets. *IEEE Trans. Software Eng.*, 8(4):371–379, 1982.

[27] Intel Code Coverage Tool in Intel C++ compiler. https://software.intel.com/en-us/node/512810.

[28] J. Kinder. Hypertesting: The case for automated testing of hyperproperties. In *3rd Workshop on Hot Issues in Security Principles and Trust (HotSpot)*, 2015.

[29] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C: A Program Analysis Perspective. *Formal Aspects of Computing Journal*, 2015.

[30] K. Ku, T. E. Hart, M. Chechik, and D. Lie. A Buffer Overflow Benchmark for Software Model Checkers. In *ASE*. ACM, 2007.

[31] J. W. Laski and B. Korel. A data flow oriented program testing strategy. *IEEE Trans. Software Eng.*, 9(3):347–354, 1983.

[32] LDRA – LDRACover. http://www.ldra.com/en/ldracover.

[33] 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.

[34] Z. Manna. *The Temporal Logic of Reactive and Concurrent Systems Specification.* Springer, 1992.

[35] P. Mathur, Aditya. *Foundations of Software Testing.* Addison-Wesley Professional, 2008.

[36] J. Myers, Glenford, C. Sandler, and T. Badgett. *The Art of Software Testing.* Wiley, 3 edition, 2011.

[37] A. J. Offutt and S. D. Lee. An empirical evaluation of weak mutation. *IEEE Trans. Software Eng.*, 20(5):337–344, 1994.

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

[39] Parasoft C/C++test: Comprehensive dev. testing tool for C/C++. https://www.parasoft.com/product/cpptest/.

[40] 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.*, 16(9):965–979, Sept. 1990.

[41] S. Rayadurgam and M. P. Heimdahl. Generating mc/dc adequate test sequences through model checking. In *Proceedings of the 28th Annual IEEE/NASA Software Engineering Workshop – SEW-03*, Greenbelt, Maryland, December 2003.

[42] I. Schieferdecker, Z. R. Dai, J. Grabowski, and A. Rennoch. The uml 2.0 testing profile and its relation to ttcn-3. In *Proceedings of the 15th IFIP International Conference on Testing of Communicating Systems*, TestCom'03, pages 79–94, Berlin, Heidelberg, 2003. Springer-Verlag.

[43] Semantic designs: C test coverage tool. http://semanticdesigns.com/Products/TestCoverage/CTestCoverage.htm.

[44] K.-C. Tai. Theory of fault-based predicate testing for computer programs. *IEEE Trans. Softw. Eng.*, 22(8):552–562, Aug. 1996.

[45] Testworks: TCAT C/C++. http://www.testworks.com/Products/Coverage/tcat.html.

[46] Testwell CTC++: Test coverage analyzer for C/C++. http://www.testwell.fi/ctcdesc.html.

[47] S. A. Vilkomir and J. P. Bowen. *Formal Methods and Testing: An Outcome of the FORTEST Network, Revised Selected Papers*, chapter From MC/DC to RC/DC: Formalization and Analysis of Control-Flow Testing Criteria, pages 240–270. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[48] 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.

[49] Q. Yang, J. J. Li, and D. M. Weiss. A survey of coverage-based testing tools. *The Computer Journal*, 52(5):589–597, 2009.

[50] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4), 1997.