# Generic and Effective Specification of Structural Test Objectives*

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

***Abstract*—A large amount of research has been carried out to automate white-box testing. While a wide range of different and sometimes heterogeneous code-coverage 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 full mutations. Besides specification, HTOL is appealing in the context of test automation as it allows handling criteria in a unified way.**

## 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. Testing all the possible program inputs being 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 an appropriate set of test inputs. 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) or all branches in the code (decision coverage). These requirements are essential to an automated white-box testing process, as they are used by testing tools 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).

**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 are restricted to a very small subset of criteria 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.*

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

**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 evaluation, test generation and infeasible objective detection.

**Proposal.** We introduce *hyperlabels*, a generic specification language for white-box test requirements. Technically, hyperlabels are a major extension of *labels* previously proposed by our team [8]. While labels can express a large range of criteria [8] (including a large part **WM'** of weak mutations **WM** [9], and a weak variant of **MCDC** [10]), they are still too limited in terms of expressiveness. For instance, labels cannot express strong variants of **MCDC** [7] or most path and 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 [14].

**Contribution.** The three main contributions of this paper are: **1.** We introduce a *novel taxonomy of coverage criteria* (Section III), orthogonal to both the standard classification [3] and the one by Ammann and Offutt [4]. Our classification is *semantic*, based on the nature of the reachability constraints

underlying a given criterion. This view is sufficient for classifying all existing criteria but full 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.*

**2.** We propose HTOL (Hyperlabel Test Objective Language), a formal specification language for test objectives (Section IV) 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 the HTOL syntax and give a formal semantics in terms of coverage. Finally, we give a few encodings of criteria beyond labels.

**3.** As a first application of hyperlabels, and in order to demonstrate their expressiveness, we provide in Section V a list of encodings for *almost all code coverage criteria defined in the Ammann and Offutt book [4]*, including many criteria beyond labels (cf. Fig. 6). The only missing criteria are strong mutations and full weak mutations, yet a large subset of weak mutations can be encoded [8].

**Potential impact.** Hyperlabels provide a *lingua franca* for defining, extending and comparing criteria in a clearly documented way. It is also a specification language for writing universal, extensible and interoperable testing tools, as we already demonstrated in practice within the LTest tool [15], [16], [14]. 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.

## II. 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 $D \triangleq D_1 \times \cdots \times D_m$, 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 $P$ ($\approx$ 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 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 $t \rightsquigarrow^k_P (loc, s)$, if $P(t)$ has the form $\sigma \cdot (loc, s) \cdot \rho$ where $\sigma$ is a partial run of length $k$. When focusing on reachability, we omit $k$ and write $t \rightsquigarrow_P (loc, s)$.

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. A wide variety of criteria have been proposed in the literature [2], [4], [3]. For example, test objectives derived from the Decision Coverage (**DC**) 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 $t \rightsquigarrow_P \mathbf{c}$ if $t$ reaches a $(loc, S)$ such that `cond` evaluates to *true* (resp. *false*) in $S$.

### B. 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* $\ell \in \mathsf{Labs}_P$ is a pair $(loc, \varphi)$ 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* $\ell \triangleq (loc, \varphi)$ in $P$, denoted $t \overset{\text{L}}{\rightsquigarrow}_P \ell$, if there is a state $s$ such that $t$ reaches $(loc, s)$ (i.e. $t \rightsquigarrow_P (loc, s)$) and $s$ satisfies $\varphi$. An *annotated program* is a pair $\langle P, L \rangle$ where $P$ is a program and $L \subseteq \mathsf{Labs}_P$ 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 $TS \overset{\text{L}}{\rightsquigarrow}_{\langle P,L \rangle} \mathbf{LC}$, if $TS$ covers every label of $L$ (i.e. $\forall \ell \in L : \exists t \in TS : t \overset{\text{L}}{\rightsquigarrow}_P \ell$).

**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 $\mathbf{LC}$ 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 1 illustrates the

simulation of some criteria with labels on sample code – that is, the resulting annotated code automatically produced by the corresponding annotation functions.

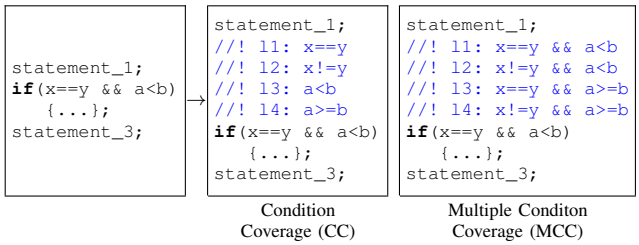


Fig. 1. Encoding of standard test requirements with labels (from [8])

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 criteria defined in [4] (except **WM** and strong mutations **SM**). The proposed extension should preserve the automation capabilities of labels.

## III. A New Taxonomy: The Cube

We propose a new taxonomy for code coverage criteria, based on the semantics of the associated reachability problem[1]. 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 [17]),

**Ext2** reachability constraints along a whole execution path (safety [18]),

**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 all coverage criteria (but full mutations) can be arranged on one of the cube vertices, 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 IV-B).

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

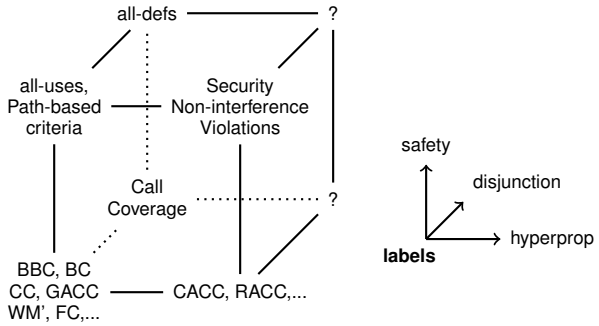


Fig. 2. The "cube" taxonomy of coverage criteria

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 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 use a disjunction of constraints over several executions of the same program.

## IV. 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 $\ell \triangleright \{v_1 \leftarrow\!\shortmid e_1; \ldots\}$ 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 $\ell$ 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. Simple 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 IV-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:

$$l \triangleq (loc_1, d) \rhd \{c_1 \leftarrow\!\shortmid \texttt{x==y}; c_2 \leftarrow\!\shortmid \texttt{a<b}\}$$
$$l' \triangleq (loc_1, \neg d) \rhd \{c_1' \leftarrow\!\shortmid \texttt{x==y}; c_2' \leftarrow\!\shortmid \texttt{a<b}\}$$
$$h_1 \triangleq \langle l \cdot l' \mid c_1 \neq c_1' \wedge c_2 = c_2' \rangle$$
$$h_2 \triangleq \langle l \cdot l' \mid c_1 = c_1' \wedge c_2 \neq c_2' \rangle$$

$h_1$ requires that the test suite reaches $loc_1$ twice (through the $\cdot$ operator) with different values for decision $d$. Values taken by $c_1$ and $c_2$ when $loc_1$ is reached are bound (through $\rhd$) to metavariables $c_1, c_2$ (first execution) and $c_1', c_2'$ (second one). These recorded values must then 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:

$$h_3 \triangleq (loc_1, \mathit{true}) + (loc_2, \mathit{true})$$

**Example 3 (all-uses)** We illustrate now the sequence operator $\dot{\rightarrow}$. 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 $h_4 \triangleq (loc_1, \mathit{true}) \rightarrow (loc_2, \mathit{true})$ and $h_5 \triangleq (loc_1, \mathit{true}) \rightarrow (loc_3, \mathit{true})$.

**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 [17], [13]. 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 $h_6$:

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

### C. Formal Definition

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

- $\ell \triangleq (loc, \varphi) \in \mathsf{Labs}_P$ 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;
- $\phi_i$ 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 $loc_{i+1}$.
- $h, h_1, h_2 \in \mathsf{Hyps}_P$ are hyperlabels;
- $\psi$ 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:

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

| | | | |
|---|---|---|---|
| $h$ | ::= | $l$ | 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 |
| $l$ | ::= | $\ell \rhd B$ | atomic label with bindings |
| $B$ | ::= | $\{v_1 \leftarrow\!\shortmid e_1; \ldots\}$ | 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 only consider well-formed hyperlabels, as defined by the $\mathrm{wf}(\cdot)$ predicate in Figure 4.

$$\frac{\forall i,j, i \neq j \Rightarrow v_i \neq v_j}{\mathrm{wf}(\ell \rhd \{v_1 \leftarrow e_1; \ldots; v_n \leftarrow e_n\})} \qquad \frac{\mathrm{wf}(h)}{\mathrm{wf}(\langle h \mid \psi \rangle)}$$

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

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

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

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

In particular, on well-formed hyperlabels, $\mathrm{nm}$ 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 $\mathrm{nm}(h) = \mathrm{nm}(h')$.

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

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 $s$, denoted $[\![B]\!]_s$, 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 $h \in \mathsf{Hyps}_P$, denoted $TS \overset{\mathrm{H}}{\rightsquigarrow}_P h$, if there exists some environment $\mathcal{E} \in \mathsf{Envs}$ such that the pair $\langle TS, \mathcal{E} \rangle$ covers $h$, denoted $\langle TS, \mathcal{E} \rangle \overset{\mathrm{H}}{\rightsquigarrow}_P h$, defined by the inference rules of Figure 5. An *annotated program* is a pair $\langle P, H \rangle$ where $P$ is a program and $H \subseteq \mathsf{Hyps}_P$ 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 $\langle P, H \rangle$, denoted $TS \overset{\mathrm{H}}{\rightsquigarrow}_{\langle P,H \rangle}$ **HLC**, if the test suite $TS$ covers every hyperlabel from $H$ (i.e. $\forall h \in H$, $TS \overset{\mathrm{H}}{\rightsquigarrow}_P h$).

The criterion simulation introduced for labels [8] is generalized to hyperlabels. Hyperlabel coverage simulates a coverage criterion $\mathbf{C}$ 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 $\mathbf{C}$ 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 $h_{\mathrm{dnf}}$ arranged as a disjunction $h_{\mathrm{dnf}} \triangleq c_1 + \cdots + c_i + \cdots + c_n$ of *guarded conjunctions* $c_i \triangleq \langle ls_1^i \cdot \ldots \cdot ls_p^i \mid \psi(B_{ls_1^i}, \cdots, B_{ls_p^i}) \rangle$ over atomic labels or sequences. The equivalence between $h$ and $h_{\mathrm{dnf}}$ is stated as

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

The main advantage of DNF is that checking whether a test suite covers $h_{\mathrm{dnf}}$ is relatively easy: we just have to find a $c_i$ for which all atomic labels or sequences $ls_j^i$ are covered and $\psi$ holds for the corresponding bindings.

### D. Advanced Example: Data-Flow on Array Cells

Standard data-flow criteria, like **all-uses**, can be refined to consider the definition and use of single array cells. Encoding such test objectives is complex as it requires constraining dynamic information. For example, in the following code, the path from $loc_1$ to $loc_3$ is a valid definition-use-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 add bindings to the atomic labels to save the values of $i, k$ and use the guard and $\rightarrow$ operators to enforce the required relationship. Encoding for the previous example is given below, with $pc$ the current line of code:

$$l_3 \triangleq (loc_1, true) \qquad l_4 \triangleq (loc_3, true)$$

$$h_7 \triangleq \langle l_3 \rhd \{v_1 \leftarrow\!\!\shortmid\ i\} \xrightarrow{\substack{pc=loc_2 \\ \Rightarrow j \neq v_1}} l_4 \rhd \{v_2 \leftarrow\!\!\shortmid\ k\} \mid v_1 = v_2 \rangle$$

## V. 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 Fig. 6, where we also specify which criteria can be expressed by atomic labels alone, and the required hyperlabel operators otherwise.

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 hyperlabels is both *necessary* and (almost) *sufficient* to encode state-of-the-art coverage criteria. *Detailed encodings are available on the companion website*[2].

## VI. Related Work

**The FQL language.** The *Fshell Query Language* (FQL) by Holzer *et al.* [11] for test suite specification represents 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. 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,

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

$$\textsc{Label}\quad \frac{t \in TS \qquad t \rightsquigarrow_P^k \langle loc, s\rangle \qquad s \vDash \varphi \qquad \mathcal{E} \supseteq [\![B]\!]_s}{t \rightsquigarrow_{\mathcal{E}}^k \langle loc, \varphi\rangle \rhd B \qquad \langle TS, \mathcal{E}\rangle \overset{\text{H}}{\rightsquigarrow}_P \langle loc, \varphi\rangle \rhd B}$$

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

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

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

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

$$\textsc{Sequence}\quad \frac{\begin{array}{c} t \in TS \qquad \forall i \in [1,n]\,,\ t \rightsquigarrow_{\mathcal{E}}^{k_i} l_i \qquad \forall i \in [1,n-1]\,,\ k_i < k_{i+1} \\ \forall i \in [1,n-1]\,,\ \forall j \in ]k_i, k_{i+1}[\,,\ (loc_j, s_j) = P(t)_j \wedge \phi_i(\mathcal{E}, loc_j, s_j) \end{array}}{\langle TS, \mathcal{E}\rangle \overset{\text{H}}{\rightsquigarrow}_P [l_1 \xrightarrow{\phi_1} l_2 \xrightarrow{\phi_2} \ldots \xrightarrow{\phi_{n-1}} l_n]}$$

Naming convention: $TS$ test suite; $\mathcal{E}$ hyperlabel environment; $h, h_1, h_2$ hyperlabels; $\psi$ hyperlabel guard predicate; $n$ positive integer ($n > 1$); $l_1, \ldots, l_n$ atomic labels with bindings; $t$ test datum; $k, k_1, \ldots, k_n$ 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 $t$; $\phi_1, \ldots, \phi_{n-1}$ predicates over sequences of labels; $\varphi$ label predicate; $B$ hyperlabel bindings.

Fig. 5. Inference rules for hyperlabel semantics

| | labels | Encodable by hyperlabels using $\xrightarrow{\phi}$ | $\|\psi\rangle$ | $\cdot$ | $+$ | See section or reference or website |
|---|---|---|---|---|---|---|
| **Control-flow graph coverage** | | | | | | |
| Statement, Basic-Block, Branch | ✓ | | | | | [8] |
| *Path coverage:* | | | | | | |
| EPC, PPC, CRTC, CPC, SPC | | • | | | | website |
| Simple Round Trip coverage | | • | | | • | website |
| **Call-graph coverage** | | | | | | |
| Function coverage (all nodes) | ✓ | | | | | [8] |
| Call coverage (all edges) | | | | | • | IV-B |
| **Data-flow coverage** | | | | | | |
| All Definitions (all-defs) | | • | | | • | website |
| + array cell definitions | | • | • | | • | as in IV-D |
| All Uses (all-uses) | | • | | | | IV-B |
| + array cell definitions | | • | • | | | IV-D |
| All Def-Use Paths (all-du-paths) | | • | | | | website |
| + array cell definitions | | • | • | | | as in IV-D |
| **Logic expression coverage** | | | | | | |
| BBC, CC, DCC, MCC | ✓ | | | | | [8] |
| *MCDC variants*: | | | | | | |
| GACC, GICC | ✓ | | | | | [8],website |
| CACC, RACC, RICC | | | • | • | | IV-B,website |
| *DNF-based criteria*: | | | | | | |
| IC, UTPC | ✓ | | | | | website |
| CUTPNFPPC | | | • | • | | website |
| **Mutation coverage** | | | | | | |
| Side-effect-free Weak Mut. | ✓ | | | | | [8] |
| (Full) Weak Mut., Strong Mut. | not encodable | | | | | |

✓: expressible by atomic labels •: required hyperlabel operators

Fig. 6. Simulation of criteria from [4]

FQL offers the ability to encode, in a 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).

**HyperLTL and HyperCTL*.** Hyperproperties [17] are properties over several traces of a system. Clarkson *et al.* [13] have introduced 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 languages could be used 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.

## VII. 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. As a first application, a universal testing tool relying on HTOL is proposed in [14]. Future work includes an 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, 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] M. Marcozzi, S. Bardin, M. Delahaye, N. Kosmatov, and V. Prevosto, "Taming Coverage Criteria Heterogeneity with LTest" in *ICST*, 2017.

[15] S. Bardin, O. Chebaro, M. Delahaye, and N. Kosmatov, "An all-in-one toolkit for automated white-box testing," in *TAP*. Springer, 2014.

[16] 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.

[17] M. R. Clarkson and F. B. Schneider, "Hyperproperties," *J. Comput. Secur.*, vol. 18, no. 6, 2010.

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