
This version is available at: https://eprints.mdx.ac.uk/5270/

Copyright:

Middlesex University Research Repository makes the University's research available electronically. Copyright and moral rights to this work are retained by the author and/or other copyright owners unless otherwise stated. The work is supplied on the understanding that any use for commercial gain is strictly forbidden. A copy may be downloaded for personal, non-commercial, research or study without prior permission and without charge.

Works, including theses and research projects, may not be reproduced in any format or medium, or extensive quotations taken from them, or their content changed in any way, without first obtaining permission in writing from the copyright holder(s). They may not be sold or exploited commercially in any format or medium without the prior written permission of the copyright holder(s).

Full bibliographic details must be given when referring to, or quoting from full items including the author’s name, the title of the work, publication details where relevant (place, publisher, date), pagination, and for theses or dissertations the awarding institution, the degree type awarded, and the date of the award.

If you believe that any material held in the repository infringes copyright law, please contact the Repository Team at Middlesex University via the following email address:

eprints@mdx.ac.uk

The item will be removed from the repository while any claim is being investigated.

See also repository copyright: re-use policy: http://eprints.mdx.ac.uk/policies.html#copy
A Formal Analysis of Requirements-Based Testing

Charles Pecheur
Université catholique de Louvain
Louvain la Neuve, Belgium
charles.pecheur@uclouvain.be

Franco Raimondi
University College London
London, UK
f.raimondi@cs.ucl.ac.uk

Guillaume Brat
RIACS - NASA Ames
Mountain View (CA), USA
guillaume.p.brat@nasa.gov

ABSTRACT
The aim of requirements-based testing is to generate test cases from a set of requirements for a given system or piece of software. In this paper we propose a formal semantics for the generation of test cases from requirements by revising and extending the results presented in previous works (e.g. [21, 20, 13]). We give a syntactic characterisation of our method, defined inductively over the syntax of LTL formulae, and prove that this characterisation is sound and complete, given some restrictions on the formulae that can be used to encode requirements. We provide various examples to show the applicability of our approach.

Categories and Subject Descriptors
D.2.5 [Software Engineering]: Testing and Debugging;
D.2.4 [Software Engineering]: Software/Program Verification

General Terms
Verification

Keywords
Coverage metrics, Requirements-based testing

1. INTRODUCTION
A number of systems currently deployed present a significant amount of complexity, as in the case of the NASA rovers Spirit and Opportunity [18, 19] exploring the surface of Mars since January 2004. The complexity of these systems make them prone to errors and there is a growing interest in tools and methodologies to perform formal verification of these systems in order to avoid safety issues, economical losses, and mission failures. For instance, in the case of the rovers,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ACM 978-1-60558-338-8/09/07 ...$10.00.

a number of conditions are imposed to avoid damage and to minimize the risk of failures; examples of conditions (i.e., requirements) include “all scientific instruments must be appropriately stored when the rover is moving” and “if the rover is at a given rock, then it must send a picture or a chemical analysis of the rock”. These kinds of conditions are called flight rules and affect various stages of system development, from design to deployment.

Typically, a model is available for this kind of applications, in the form of a labelled transition system or some other equivalent formalism (e.g. a Promela model for the model checker SPIN [12], a planning model written in PDDL [10], etc.). The availability of such models makes theoretically possible the direct verification of flight rules in a formal way using model checking and verifying the requirement against the given model. In practice, a number of issues arise:

• the size of the state space may be too large to be analysed exhaustively with a model checker;

• the model of the system could be provided in a language that is not easily encoded in the input language of a model checker (see, for instance, the problem of translating PDDL models into an adequate input for a model checker [15, 14]);

• consider the formula “if the rover is moving, then all instruments are stored”: this formula could be true because the rover never moves, which is something a model checker cannot capture directly. In some cases, we are interested in “stressing” a particular atomic proposition in a formula, and make a formula true because of that particular proposition.

As a consequence, testing comes as a natural choice to enable the verification of domains that cannot be translated into model checking problems for the first two issues mentioned above. Moreover, the third issue can be alleviated by extending the Modified Condition/Decision Coverage (MC/DC) metric: this metric is required by critical avionic software and can be used to stress all the atoms in a formula (see Section 3.1 for further details).

However, MC/DC only reasons about propositional formulae appearing in a program; instead, our concern here is to reason about requirements and provide a metric for the coverage of flight rules usually expressed in temporal logic. Intuitively, a requirement (expressed as a temporal logic formula) is covered by a set of executions paths of the system. Our aims in this paper are twofold:
1. Define in a formal way what it means for an execution
path \( \pi \) to be an adequate test case for a formula \( \varphi \)
and an atom \( a \) appearing in the formula. We will use
the notation \( \text{FLIP}(\varphi, a) \) to denote the set of execution
paths that are adequate tests for \( a \) in \( \varphi \) (the meaning
of FLIP will become clear in the following sections).

2. Given a formula \( \varphi \) and an atom \( a \), provide a procedure
to derive a new formula \([\varphi]_a\) (called a trap formula)
such that
\[
\pi \models [\varphi]_a \iff \pi \in \text{FLIP}(\varphi, a)
\]
i.e., the test cases for a formula \( \varphi \) being true in a path
\( \pi \) because of atom \( a \) are all (and only) the paths \( \pi \) that
satisfy the trap formula \([\varphi]_a\).

Recent works in this direction include [21, 20, 13]: a discussion
and comparison is presented in Section 3.2.

The rest of the paper is organised as follows: we introduce
our notation and preliminary concepts in Section 2. In Section
3 we start by reviewing Modified Condition/Decision
Coverage (MC/DC), a metric for Boolean expressions in a
program; we then review related work, and we introduce
our metric FLIP in Section 3.3, proving the correctness of
our metric. We provide examples of our metric in Section
4, together with a discussion on how results should be
interpreted. We conclude in Section 5.

2. PRELIMINARIES AND NOTATION

We assume some familiarity with temporal logic and with
LTL in particular. We refer to [6] for more details.

Consider a LTL formula \( \varphi \), interpreted over (finite or infin-
ite) paths \( \pi \), built from a set of states \( S \). By a slight abuse
of notation, we equate logic formulae to their semantic val-
idity set and consider that
\[
\varphi(\pi) \equiv \pi \models \varphi \equiv \pi \in \varphi
\]
Given a finite path \( \pi = s_1 \rightarrow s_2 \rightarrow \cdots \rightarrow s_n \) in \( S^n \) and
\( 1 \leq i \leq j \leq |\pi| \), we define
\[
\pi(i) = s_i
\]
\[
\pi(i : j) = s_i \rightarrow \cdots \rightarrow s_j
\]
\[
\pi(i :) = s_i \rightarrow \cdots
\]
By extension, for infinite paths \( \pi = s_1 \rightarrow s_2 \rightarrow \cdots \) in \( S^\omega \),
\( |\pi| = \omega \) and these notations still apply, with \( \pi(i :) = s_i \rightarrow \cdots \).

2.1 Projections and Variants

Let \( AC(\varphi) \) be the set of atomic conditions in a formula
\( \varphi \), and \( a \in AC(\varphi) \) one such condition. We write \( s(a) \)
for the truth value of condition \( a \) in state \( s \), and \( \pi(a) \) for the
sequence of truth values of \( a \) along states of a path \( \pi \). More
generally, given a set of atomic conditions \( X \) we denote by
\( s(X) \) the (vector of) values of conditions in \( X \) in state \( s \) and
by \( \pi(X) \) the sequence of such values along \( \pi \) (also called the
projection of \( \pi \) over \( X \)).

Definition 1. Given \( X \subseteq AC(\varphi) \), a path \( \pi' \) is an \( X \)-
variant of a path \( \pi \), denoted \( \pi \sim_X \pi' \), if
\[
\pi(AC(\varphi) - X) = \pi'(AC(\varphi) - X)
\]

In what follows, we will only consider variants with re-
spect to a single condition \( a \), which we will denote \( \pi \sim_a \pi' \).
Obviously, \( \sim_a \) is an equivalence relation over paths, so each
path \( \pi \) induces an equivalence class \([\pi]_a = \{ \pi' \mid \pi \sim_a \pi' \}\).
By construction, if \( \pi \sim_a \pi' \), then \( |\pi| = |\pi'| \) and for any \( i, j \) we have that \( \pi(i : j) \sim_a \pi'(i : j) \).

2.2 Linearity

This section discusses the distinction between atomic condi-
tions \( a_1, \ldots, a_n \) occurring in a (propositional or temporal)
formula \( \varphi(a_1, \ldots, a_n) \) and the possibly multiple occurrences
of the same condition, and the impact on the functional
dependency between the value of those conditions and the
value of the formula. This will be used to consider coverage
of test cases with respect to single occurrences (or possibly
multiple covariant occurrences, see below) of a given condi-
tion.

Definition 2. A formula \( \varphi \) is linear in a condition \( a \) iff \( a \)
occurrs only once in \( \varphi \). It is constant in \( a \) if \( a \) does not occur
in \( \varphi \).

For instance, \( F (a \land b) \land F (\neg a \land c) \) is linear in \( b \) and \( c \) but
it is not linear in \( a \).

Let \( count(a, \varphi) \) be the number of occurrences of \( a \) in \( \varphi \).
\( \varphi \) is linear or constant in \( a \) iff \( count(a, \varphi) = 1 \) or 0. The
linearisation of \( \varphi \), denoted \( lin(\varphi) \) is obtained by replacing
every occurrence of any condition \( a \) in \( \varphi \) by a distinct variant
\( a' \), where \( 1 \leq i \leq count(a, \varphi) \). By construction, \( lin(\varphi) \) is
linear in all its conditions. We define \( unlin(lin(\varphi)) = \varphi \); in
particular \( unlin(a') = a \).

In what follows, we will restrict the analysis to formulae
that are linear in their conditions, or equivalently, consider
multiple occurrences of the same condition in a formula as
distinct conditions.

2.3 Monotonicity

We define an ordering relation on paths, based on the
value of a condition \( a \). Given two paths \( \pi, \pi' \) such that
\( |\pi| = |\pi'| \),
\[
\pi \sqsubseteq_a \pi' \iff \pi \sim_a \pi' \land \forall i \leq |\pi| \cdot \pi(i)(a) = \pi'(i)(a)
\]
where \( \pi(i)(a) \) denotes the value of \( a \) in the \( i \)-th state of \( \pi \).
Intuitively, this means that \( a \) is true “more often” over \( \pi' \)
than over \( \pi \), all other conditions remaining the same.

It is easily seen that \( \sqsubseteq_a \) is a Boolean lattice over each
equivalence class \([\pi]_a \), with Boolean operations \( \land_a, \lor, \neg_a \)
defined point-wise, e.g. \( \pi' \land_a \pi'' \) is a new path \( \pi \) where
\( \pi(i)(a) = \pi'(i)(a) \land \pi''(i)(a) \). Top and bottom elements of
this lattice are \( \pi[a := T] \) and \( \pi[a := F] \), where \( \pi[a := v] \)
means the \( a \)-variant of \( \pi \) where \( \pi(i)(a) = v \) for all indices \( i \).

Definition 3. Given a condition \( a \), a temporal formula \( \varphi \)
is covariant (resp. contravariant) in \( a \) iff for all \( \pi \sqsubseteq_a \pi' \)
(resp. \( \pi' \sqsubseteq_a \pi \)) we have that if \( \pi \models \varphi \) then \( \pi' \models \varphi \).
\( \varphi \) is monotonic in \( a \) iff it is either covariant or contravariant in
\( a \).

All usual logic operators preserve monotonicity, in the fol-
lowing sense:

\[1\] The notation \([\pi]_a \) over paths is not to be confused with
forthcoming notation \([\varphi]_a \) for trap formulae.
Definition 4. Given a condition \( a \), a temporal logic operator \( \theta \) is covariant (resp. contravariant) in \( a \) if for any \( \varphi \) that is covariant in \( a \), \( \theta(\varphi) \) is covariant (resp. contravariant) in \( a \), and vice-versa for \( \varphi \) contravariant in \( a \). \( \theta \) is monotonic in \( a \) if it is either covariant or contravariant in \( a \).

A composition \( \theta' \circ \theta \) is monotonic if \( \theta \) and \( \theta' \) are both monotonic, where the usual sign rules apply for variance. As base cases, \( a \) and \( \neg a \) are obviously covariant and contravariant in \( a \). If \( a \) does not occur in \( \varphi' \), then \( \bullet \land \varphi' \) and \( \bullet \lor \varphi' \) are covariant and contravariant in \( a \). Note however that \( \odot \) and \( \equiv \) are not monotonic. All LTL operators (\( X, U, R, F, G \)) are covariant in \( a \) with respect to each of their arguments, if the other argument is constant in \( a \). This is because (i) these operators are covariant in their arguments in the logical sense (if \( \varphi \Rightarrow \varphi' \) then \( \theta(\varphi) \Rightarrow \theta(\varphi') \)) and (ii) \( \pi \models \varphi_a \lor \varphi' \) (or any other temporal operator) depends only on \( \pi(i : ) \models \varphi_a \) and \( \pi(j : ) \models \varphi' \) on suffixes of \( \pi \), and \( \pi \models \varphi \)' implies \( \pi(i : ) \sqsubseteq \varphi' \).\(^3\) As a result, if a formula \( \varphi \) (built from these operators) is linear in \( a \) then it is monotonic in \( a \).\(^4\)

3. COVERAGE OF REQUIREMENTS

In this section we present our approach to automatically generate test cases from a requirement expressed in LTL. We begin by briefly reviewing MC/DC coverage (Section 3.1 and previous approaches (Section 3.2)), and we introduce our approach in Section 3.3.

3.1 Overview of MC/DC coverage

MC/DC coverage is required for the most critical categories of avionic software [17] and it is defined in terms of the Boolean decisions in the program, such as test expressions in if and while statements, and the elementary conditions (i.e. Boolean terms) that compose them. A test suite is said to achieve MC/DC if its execution ensures that:

1. Every basic condition in any decision has taken on all possible outcomes at least once.
2. Each basic condition has been shown to independently affect the decision’s outcome.

As an example, the program fragment if \( (a \land b) \{ \ldots \} \) contains the decision \( c \equiv (a \lor b) \) with conditions \( a \) and \( b \). MC/DC is achieved if this decision is exercised with the following three valuations:\(^2\)

\[
\begin{array}{ccc}
 a & b & a \lor b \\
 T & T & T \\
 T & F & T \\
 F & T & T \\
 F & F & T \\
\end{array}
\]

Indeed, evaluations 1 and 3 only differ in \( a \), showing cases where \( a \) independently affects the outcome of \( c \), respectively in a positive and negative way. The same argument applies to evaluations 2 and 3 for \( b \). In particular, if \( a = T \), then \( a \) affect \( \varphi \) positively, and if \( a = F \), then \( a \) affect \( \varphi \) negatively.\(^3\)

There is some flexibility in how “independently affect” is to be interpreted, see [4, 11, 3]. The original definition in [17] requires that each occurrence of a Boolean atom be treated as a distinct condition, and that independent effect be demonstrated by varying that condition only while keeping all others constant. This makes it difficult or impossible to achieve MC/DC if there is a coupling between conditions in the same decision, and in particular if the same atom occurs several times (e.g. \( a \) in \( (a \land b) \lor (\neg a \land c) \)).

Several variants have been proposed and defined to address that problem. The original definition is known as unique cause MC/DC, while [11] defines a weaker version based on logic gate networks, called masking MC/DC.

3.2 Related work

Coverage metrics for temporal logic have been presented in the past. In [13], a metric is provided to measure the degree of coverage of a model. This is fundamentally different from what we do in this paper, as our aim is to cover requirements expressed in LTL.

A metric for specifications is provided in [20] using mutations: intuitively, a “good” test case is a path that can “detect” all mutations of a formula. The authors also define the notion of vacuous sub-formula, which present an interesting similarity with our notion of \( a \)-variants presented above. In this approach tests are generated using a model checker, and thus differ from our work in that we provide a constructive method starting from formulae directly.

The work presented in [21] shares most of our aims. The notion of unique first cause is defined in [21] as \( a \) condition \( c \) is the unique first cause (UFC) for \( \varphi \) along a path \( \pi \) if, in the first state along \( \pi \) in which \( \varphi \) is satisfied, it is satisfied because of \( a \). This definition is a generalization of the notion of MC/DC presented above.

A syntactic characterization of UFC is proposed in [21]: for a formula \( \varphi \) and condition \( a \) in \( \varphi \), a trap formula \( \text{ufc}(\varphi, a) \) can be derived such that \( \text{ufc}(\varphi, a) \) holds in all suitable execution paths in which \( a \) is the first cause for \( \varphi \). We report here only a few derivation rules that will be used below and refer to [21] for further details (\( \varphi_a \) denotes a formula in which \( a \) occurs):

\[
\begin{align*}
\text{ufc}(\varphi, a) &= F & \text{where } a \text{ does not occur in } \varphi \\
\text{ufc}(a, a) &= a \\
\text{ufc}(\varphi_a \lor \psi) &= \text{ufc}(\varphi_a, a) \land \neg\psi \\
\text{ufc}(F \varphi_a, a) &= \neg\varphi_a \lor \text{ufc}(\varphi_a, a)
\end{align*}
\]

These definitions are further refined in a number of ways in [21] to deal with finite/truncated execution paths, but no formal proof is provided that the definition of \( \text{ufc}(\varphi, a) \) corresponds indeed to the fact that “\( \varphi \) is true because of \( a \)” and, moreover, the causality link in the definition is not formally defined and may be subject to ambiguous interpretations.

As a running example, consider the formula \( \varphi = F(a \land b) \). The derivation rules presented above give

\[
\text{ufc}(\varphi_a, a) = (\neg a \land \neg b) \lor (a \land \neg b)
\]

The trace \( \tau = \{ 1 \} \rightarrow \{ a \} \rightarrow \{ \} \) does satisfy \( \text{ufc}(\varphi, a) \). However, this trace does not guarantee that \( a \) is the unique cause for \( a \). Indeed, it is not possible to flip the value of \( a \) in any way to make \( F(a \land b) \) false along this trace (cf. the criteria
appearing in MC/DC). In essence, using the definition of α-variant defined above, there is no α-variant of π such that ϕ does not hold along this variant.

Other examples can be found for the remaining operators: the essence is that ufc(ϕ, a) is satisfied along all valid test cases, but it is also satisfied on execution paths that are not adequate test cases, i.e., ufc(ϕ, a) is “too generous”. Equivalent, as it will be described below, ufc(ϕ, a) is not sound with respect to the definition of adequate test cases.

### 3.3 FLIP

In this section we introduce our definition of “adequate” test cases and we present a syntactic derivation of trap formulae for adequate test cases. We also prove the soundness and correctness of our approach.

**Definition 5.** An execution path π is an adequate test case for an atom a occurring in a formula ϕ if π ⊨ ϕ and there exists an α-variant π′ of π such that π′ ▷ ϕ. We denote with FLIP(ϕ, a) the set of all such paths.

We denote with [ϕ]a the trap formula characterising adequate test cases (i.e., test cases as defined in Definition 5). We define [ϕ]a by structural induction on LTL formulae in negation normal form as follows:

**Definition 6.** Syntactic characterisation of trap formulae

\[
\begin{align*}
[ϕ]_a &= F \quad \text{where} \ a \ \text{does not occur in} \ ϕ' \\
[a]_a &= a \\
[¬a]_a &= ¬a \\
[ϕ ∧ ϕ']_a &= [ϕ]_a ∧ [ϕ']_a \\
[ϕ ∨ ϕ']_a &= [ϕ]_a ∨ [ϕ']_a \\
[X ϕ]_a &= X[ϕ]_a \\
[ϕ U ϕ']_a &= (ϕ' U ϕ) ∧ (¬ϕ' R [ϕ]_a) \\
[ϕ U ϕ'']_a &= (ϕ' U ϕ) ∧ (¬ϕ' U ([ϕ]_a ∧ ¬ϕ')) \\
[ϕ R ϕ']_a &= (ϕ R ϕ') ∧ ((ϕ ∨ [ϕ]_a) U ¬ϕ') \\
[ϕ' R ϕ'']_a &= (ϕ' R ϕ') ∧ (¬ϕ' U [ϕ]_a) \\
\end{align*}
\]

(where R is the standard release operator).

Other cases are obtained by syntactic derivation:

\[
\begin{align*}
[F ϕ]_a &= F ϕ ∧ G([ϕ]_a) \\
[G ϕ]_a &= G ϕ ∧ F([ϕ]_a) \\
[ϕ' W ϕ]_a &= (ϕ' W ϕ) ∧ ((ϕ ∨ [ϕ]_a)) \\
U(¬ϕ ∧ [ϕ]_a) &= (ϕ U [ϕ]_a) \\
[ϕ ϕ'']_a &= (ϕ W ϕ') ∧ (¬ϕ U (¬ϕ' ∧ [ϕ]_a)) \\
\end{align*}
\]

These derivations for fixed point modalities are all of the form [ϕ]a = ϕ ∧ ϕ′′, where the recursive step occurs only in ϕ′′. They can be rewritten into equivalent forms [ϕ]a = ϕ1 U (([ϕ]a ∧ ϕ2):

\[
\begin{align*}
[ϕ' U ϕ]_a &= (ϕ' ∧ ¬[ϕ]_a) U ([ϕ]_a ∧ ¬ϕ' R [ϕ]_a) \\
[ϕ U ϕ']_a &= (ϕ ∧ ¬ϕ') U ([ϕ]_a ∧ ¬ϕ' R [ϕ]_a) \\
[ϕ R ϕ']_a &= (¬ϕ ∧ ϕ') U ([ϕ]_a ∧ ϕ ∧ [ϕ]_a) \\
[ϕ' R ϕ']_a &= (¬ϕ ∧ ϕ') U ([ϕ]_a ∧ ϕ ∧ [ϕ]_a) \\
[F ϕ]_a &= ¬[ϕ]_a U ([ϕ]_a ∧ G [ϕ]_a) \\
[G ϕ]_a &= [ϕ]_a U ([ϕ]_a ∧ G [ϕ]_a) \\
\end{align*}
\]

**Proof of correctness.** We now show that π ⊨ [ϕ]a if and only if π ∈ FLIP(ϕ, a). We prove completeness (only if) and soundness (if) separately.

We first need the following simple lemma:

**Theorem 1 (Extremal α-variants).** If ϕ is covariant (resp. contravariant) in a and π ∈ [ϕ]a then π[a := F] ▷ ϕ (resp. π[a := T] ▷ ϕ).

**Proof.** We prove the covariant case; the contravariant case follows by duality. Since π ∈ [ϕ]a, there is a π′ ▷ ϕ such that π′ ⊨ ϕ. By contra-position of covariance, if π′′ ⊨ ϕ and π′ ▷ ϕ then π′′ ▷ ϕ. But π[a := F] is minimal in [π]a, therefore π[a := F] ⊨ ϕ and π[a := F] ▷ ϕ. □

**Theorem 2 (Soundness of [ϕ]a).** Let ϕ be a formula in negation-normal form, linear and monotonic in a. If π ⊨ [ϕ]a, then π ⊨ ϕ and there exists π′ ▷ ϕ such that π′ ⊨ ϕ (i.e., π ∈ FLIP(ϕ, a)).

**Proof.** Let π ⊨ [ϕ]a. We have to show that π ⊨ ϕ and build π′ such that π′ ▷ ϕ and π′ ⊨ ϕ. By induction, we can assume that for any sub-formula ϕ1 of ϕ, if π1 ⊨ [ϕ1]a, then π1 ⊨ ϕ1 and there is π′1 ▷ π1 such that π′1 ⊨ ϕ1.

The proof goes by structural induction on ϕ, where ϕa is the sub-formula where a occurs. We consider the case where ϕa is covariant in a; the contravariant case follows by duality.

- [ϕ]a = F, where a does not occur in ϕ:
  Trivially, there is no such π.
- [a]a = a:
  Obviously, π ⊨ ϕ, which means π(1)(a) = T and π′ = π[a := F] ▷ ϕ.
- [¬a]a = ¬a:
  Dual of the previous case: π(1)(a) = F and π′ = π[a := T] ▷ ϕ.
- [ϕ ∧ ϕ′]a = [ϕ]a ∧ [ϕ′]a:
  Since π ⊨ [ϕ]a, π ⊨ ϕa and there is π′ ▷ ϕ such that π′ ▷ ϕa (and a does not occur in ϕ′). Hence π ⊨ ϕ ∧ ϕ′ and π′ ▷ ϕ ∧ ϕ′.
- [ϕ ∨ ϕ′]a = [ϕ]a ∧ [ϕ′]a:
  Similar to ϕ ∧ ϕ′, but in this case π ▷ ϕ′ so π′ ▷ ϕ′.
- [X ϕ]a = X[ϕ]a:
  We have π(2) := [ϕ]a so (i) π(2) := ϕa and thus π ⊨ ϕ and (ii) there is π′2 ▷ π(2) such that π′2 ▷ ϕa.
  Build π′(1) = π(1) and π′(2) = π′2, thus π′ ▷ ϕ.
- [ϕ U ϕ']a = (ϕ' U ϕ) ∧ (¬ϕ' R [ϕ]a):
  Obviously π ⊨ ϕ.
  Let π the first index such that π(n) ▷ ϕ′ (n may be infinite). Consider all j such that j ≤ n π(j) ▷ ϕa. Because π = (¬ϕ R [ϕ]a), we have π(j) := [ϕ]a. Let π′ = π[a := F].
  By monotonicity of ϕa (theorem 1), π′(j) := ϕa for all j ≤ n. Therefore there is no j ≤ n such that π′(j) := ϕa, so π′ ▷ ϕ.
Obviously \( \pi \models \psi \). Since \( \pi \models (\varphi_a \land \lnot \varphi') \), there is a minimal finite such that \( \pi(n) \models \lnot \varphi' \). The proof follows similarly as for \( \varphi' \land \varphi_a \), with the difference that the finite \( n \) guarantees that \( \varphi \land \varphi' \) is indeed eventuality falsified along \( \pi[a := \text{F}] \).

- \([\varphi \land \varphi']_a = (\varphi' \land \varphi_a) \land (\varphi \land \lnot \varphi') \):
  
  We readily have \( \pi \models \varphi' \land \varphi_a \). Let \( m \) the first indices such that \( \pi(m) \models \varphi_a \) and \( \pi(n) \models \lnot \varphi' \). Because \( \pi \models \varphi' \land \varphi_a \), we have that \( m \) is finite and \( m \leq n \) (may be infinite).

For all \( i < n \) we have \( \pi'(i) \models \varphi' \), and \( \pi'(n) \models \lnot \varphi' \), since \( a \) does not occur in \( \varphi' \). Let \( m \leq j \leq n \) such that \( \pi(j) \models \varphi_a \). We have that \( \pi'(j) \models \lnot \varphi_a \), otherwise \( \pi' \models \varphi \). Hence \( \pi(j) \models [\varphi_a]_a \) by inductive hypothesis, and \( \pi \models \lnot \varphi' \land \varphi_a \). (See Figure 1 for a visual representation).

- \([\varphi \land \varphi']_a = (\varphi \land \varphi_a) \land (\varphi' \land \lnot \varphi') \):

This is similar to \( \varphi \land \varphi' \), except that \( n \) can be finite, for otherwise \( \pi \models G \varphi' \) and thus \( \pi' \models \varphi \). Also, in this case \( m < n \) and it is enough to consider all \( j < n \), since \( \pi \models \varphi \land \varphi' \) if \( m = n \).

Theorem 3 (Completeness of \([\varphi_a]_a \). Let \( \varphi \) be a formula in negation-normal form, linear and monotonic in \( a \).

If \( \pi \models \varphi \) and there exists \( \pi' \models \pi \) such that \( \pi' \not\models \varphi \) (i.e., if \( \pi \in \text{FLIP}(\varphi, a) \)), then \( \pi \models [\varphi_a]_a \).

Let \( \pi' \models \pi \) such that \( \pi \models \varphi \) and \( \pi' \not\models \varphi \). We have to show that \( \pi \models [\varphi_a]_a \). By induction, we can assume that for any sub-formula \( \varphi_a \) of \( \varphi \), if there is \( \pi_1 \models \pi \) such that \( \pi_1 \models \varphi \) and \( \pi'_1 \not\models \varphi \), then \( \pi_1 \models [\varphi_a]_a \).

- \([\varphi'_a]_a = F \)
  
  where \( a \) does not occur in \( \varphi' \):
  
  If \( a \) does not occur in \( \varphi' \) and \( \pi \models \varphi' \), then for all \( \pi' \models \pi \) we also have \( \pi' \models \varphi' \).

- \([a]_a = a \):
  
  Obviously, \( \pi \models a \).

- \([-a]_a = -a \):
  
  Obviously, \( \pi \models -a \).

- \([\varphi \land \varphi']_a = [\varphi_a]_a \land \varphi' \):
  
  We have \( \pi \models \varphi_a \) and \( \pi \models \varphi' \) so \( \pi' \models \varphi \) because \( \varphi' \) does not depend on \( a \). However \( \pi \not\models \varphi_a \land \varphi' \) so we must have \( \pi' \not\models \varphi_a \). Hence \( \pi \models [\varphi_a]_a \land \varphi' \).

- \([\varphi \lor \varphi']_a = [\varphi_a]_a \lor \varphi' \):
  
  Similar to \( \varphi_a \land \varphi' \) except \( \pi \not\models \varphi \) so \( \pi' \not\models \varphi ' \).

- \([X \varphi_a]_a = X [\varphi_a]_a \):
  
  We have \( \pi' \not\models X \varphi_a \). In general, \( X \varphi_a = \lnot \varphi_a \land \lnot X \varphi_a \). However, \( \pi ' \models \pi \) and \( \pi ' \) must have the same length: since \( \pi \models X \pi \), we also have \( \pi ' \models X \pi \) and therefore \( \pi ' \models X \varphi_a \). Hence \( \pi(2:s) \models \varphi_a \) and \( \pi '(2:s) \models -\varphi_a \), so \( \pi (2:s) \models [\varphi_a]_a \).
Figure 2: Maude definition of FLIP.

our characterisation does not allow for the problematic path
π = \{\} \to \{a\} \to \{b\} presented in Section 3.2.

We present further examples in the next section to illustrate the applicability of our approach.

4. EXAMPLES

We have implemented the rules of Definition 6 in a Maude module (Maude is an automated reasoning engine based on rewriting logic [7]). The module extends the temporal model checking module available in the standard Maude distribution. Our extension defines a new operation “square brackets”:

op [ ]_ : Formula Atom -> Formula
vars C C' : Atom .
eq [ ]_ C = C .
eq [ C']_ C = False [otherwise] .
eq [ \neg C']_ C = False [otherwise] .
ceq [ X / Y]_ C = ( [X]C \land Y) if C in X .
ceq [ X / Y] C = ( [X]C \land \neg Y) if C in X .
eq [0]X]_C = 0 [X]C .
ceq [X U Y] C = (X U Y) \land (\neg Y U ( [X]C \land Y)) if C in X .
ceq [X U Y] C = (X U Y) \land (\neg X R (Y -> [Y]C)) if C in Y .
ceq [X R Y] C = (X R Y) \land ([X]C U Y) if C in X .
ceq [X R Y] C = (X R Y) \land (\neg X U [Y]C) if C in Y .
ceq [X]C = False [otherwise] .
ceq [X]C = False [otherwise] .
eq [C] C = C .

vars X Y Z : Formula .
vars C C' C'' : Atom .

This formula characterises all the paths in which \neg MarkasUnread occurs at least once: this is guaranteed by the first part of the formula, in which it is required that PlacedinMailboxes must be false at some point in the future, followed by globally \neg MarkasUnread. Thus, this formula rules out the possibility that MarkedasUnread is always true: indeed, in this case it would be impossible to flip the value of the original formula because of MarkasUnread.

The Maude module can be used to produce the trap formula for the other atom PlacedinMailboxes as well. Each of these two trap formulae encode a set of execution paths. The coverage of the original requirement is achieved by verifying that the system allows for execution paths that belong into each of these sets. The actual verification of these inclusions is performed in different ways, depending on the kind of model under investigation. In the following section we present an application to a NuSMV model.

4.1 An example with a model and interpretation of the results

Consider the standard NuSMV [5] example of mutual exclusion of two asynchronous processes by means of a semaphore (see the code in Figure 4).

A property of this protocol is the following:

$$\varphi_{ME} = G ((\text{plc} \lor \text{p2c}) \Rightarrow F (\text{plc} \lor \text{p2c}))$$

where plc is a short-hand for proc1.state=entering, and similarly for the remaining atoms.

This property encodes the fact that if both processes are trying to enter the critical section, at least one of them will eventually enter it, and NuSMV can be used to show that \varphi_{ME} holds. We can use our Maude module to derive the trap formula encoding an adequate test case for plc (i.e., for process 1 in the critical state, see Figure 5). The trap formula can be simplified to:

$$F (\text{plc} \lor \text{p2c}) \land G (\neg\text{p2c}) \land (\text{plc} \land \text{p2e}))$$

This formula is satisfied along a path that (1) satisfies the original requirement and (2) has a state where both processes are trying to enter the critical section, and (3) nowhere along the path does the second process enter the critical section (in this way it is possible to flip the true value of the formula because of the first process entering the critical section).

By taking the negation of the formula above we can use NuSMV to check whether an execution of the system (i.e., the model of Figure 4) exists such that the atom plc can flip the value of the formula. Indeed, if such an execution exists,
The four cases are as follows:

- Describe sets of execution paths encoded by a trap formula.
- Paths enabled by the original model, and the sets (a) to (d) denote the set of maiming three trap formulae (one for each of the Boolean variables).
- The first state of the required execution path.
- This is the case with the trap formula reported above; a counterexample should be produced describing the required path. This is the expected outcome and the test succeeds (case (d) in the Figure).

Notice that there are four possible outcomes for the verification of a conditional property, e.g., the trap formula may impose constraints on the truth value of a variable. The formula \( \varphi \) is a property that should be false ("negative property"), and \( \neg \varphi \) true in the model: this is the expected outcome and the test succeeds (case (d) in the Figure).

The formula \( \varphi \) is a property that should be false ("negative property") and \( \neg \varphi \) true in the model (thus producing a counterexample): this result implies a design error (a bug): this is case (c) in the Figure, where execution paths exist for \( \varphi \) but not for \( \neg \varphi \).

Notice that \( \neg \varphi \) false implies that the original requirement is violated, too, as the trap formula can be rewritten as a conjunction of the original formula with additional constraints.

In this section we have presented an example using a NuSMV model and thus the original requirement could have been verified directly. However, this would have not guaranteed the coverage of all the atoms in the requirement.

A similar methodology can be applied to other kinds of models that cannot be verified in full. For instance, in the case of a planning model in PDDL [10], trap formulae generated with Maude can be translated into new planning goals and added to the original model as additional constraints, therefore forcing the generation of plans that satisfy the trap formulae. This kind of application is presented in more details in [2].

5. DISCUSSION AND CONCLUSION

In this paper we have presented a method to generate test cases from requirements expressed in LTL, extending the notion of MC/DC to temporal formulas in a formal setting. However, care must be taken when comparing the two metrics. Although a strong similarity can be drawn between MC/DC coverage for decisions in programs and FLIP coverage for requirements in temporal logic, the context of application, and hence the interpretation of the results, are quite different. In MC/DC coverage for a condition \( F \), both

---

\[\text{MODULE user(semaphore)}\]
\[\text{VAR}\]
\[\text{state : (idle, entering, critical, exiting);}\]
\[\text{ASSIGN}\]
\[\text{init(state) := idle;}\]
\[\text{next(state) :=}\]
\[\text{case}\]
\[\text{state = idle : (idle, entering);}\]
\[\text{state = entering & !semaphore : critical;}\]
\[\text{state = critical : (critical, exiting);}\]
\[\text{state = exiting : idle;}\]
\[\text{1 : state;}\]
\[\text{esac;}\]
\[\text{next(semaphore) :=}\]
\[\text{case}\]
\[\text{state = entering : 1;}\]
\[\text{state = exiting : 0;}\]
\[\text{1 : state;}\]
\[\text{esac;}\]

---

\[\text{MODULE main}\]
\[\text{VAR}\]
\[\text{semaphore : boolean;}\]
\[\text{proc1 : process user(semaphore);}\]
\[\text{proc2 : process user(semaphore);}\]
\[\text{ASSIGN}\]
\[\text{init(semaphore) := 0;}\]
\[\text{FAIRNESS}\]
\[\text{running}\]

---

Figure 4: NuSMV code for mutual exclusion

then the negation of the trap formula should be false, and a counterexample should be produced describing the required path. This is the case with the trap formula reported above; a screen-shot from NuSMV is reported in Figure 6 describing the first state of the required execution path.

Full coverage of \( \varphi_{ME} \) is achieved by computing the remaining three trap formulae (one for each of the Boolean atoms) and by repeating their verification with NuSMV.

Notice that there are four possible outcomes for the verification using a model checker of (the negation of) a trap formula \( \varphi \), encoding a adequate test cases for an atom \( a \) in a formula \( \varphi \). Consider Figure 7, where \( M \) denotes the set of paths enabled by the original model, and the sets (a) to (d) describe sets of execution paths encoded by a trap formula. The four cases are as follows:

- The formula \( \varphi \) is a desired ("positive" property), and \( \neg \varphi \) is true in the model (and a counterexample is produced by NuSMV): this is case (a), where the negation of the trap formula is false in the intersection of \( M \) and (a), and it is possible to guarantee that \( \varphi \) has been covered with respect to \( a \) by using one of the paths in this intersection. This is the case for the example formula presented above for the mutual exclusion protocol and NuSMV produces one of the paths as a counter-example.

- The formula \( \varphi \) is a desired ("positive" property) and \( \neg \varphi \) true in the model (case (b) in the figure: notice that the set depicted is the trap formula, thus its negation includes all the possible executions \( M \) of the model). This means that the test for \( a \) in \( \varphi \) fails (i.e., it is not possible to cover \( a \) in \( \varphi \) using the model). The trap formula might not be exercised for a number of reasons, e.g., the trap formula may impose constraints on the model that are inconsistent with the original model. One possibility is that an atom \( a \) in a formula

Figure 5: Maude screen-shot for the mutual exclusion example

could be coupled to another atom \( b \) so that it is not possible to change the value of \( a \) without affecting \( b \). As an example consider \( a \) to encode the proposition \( x \leq 0 \) and \( b \) to encode \( x > 0 \): in this case a change in the truth value of \( a \) causes a change in \( b \). In general, if a "positive" case fails, further investigations are needed and the result does not imply a system failure (in fact, couplings such as \( a \) and \( b \) above may well be required).

- The formula \( \varphi \) is a property that should be false ("negative property"), and \( \neg \varphi \) true in the model: this is the expected outcome and the test succeeds (case (d) in the Figure).

- The formula \( \varphi \) is a property that should be false ("negative property") and \( \neg \varphi \) false in the model (thus producing a counterexample): this result implies a design error (a bug): this is case (c) in the Figure, where execution paths exist for \( \varphi \) but not for \( \neg \varphi \).

Notice that \( \neg \varphi \) false implies that the original requirement is violated, too, as the trap formula can be rewritten as a conjunction of the original formula with additional constraints.

5. DISCUSSION AND CONCLUSION

In this paper we have presented a method to generate test cases from requirements expressed in LTL, extending the notion of MC/DC to temporal formulas in a formal setting. However, care must be taken when comparing the two metrics. Although a strong similarity can be drawn between MC/DC coverage for decisions in programs and FLIP coverage for requirements in temporal logic, the context of application, and hence the interpretation of the results, are quite different. In MC/DC coverage for a condition \( F \), both
Figure 6: NuSMV counterexample for a trap formula.

\[ F = T \text{ and } F = F \] correspond to alternate paths in the program, that one is a priori equally interested in covering. In FLIP coverage of a requirement \( \varphi \), however, the situation is strongly asymmetrical: the case where \( \pi \models \varphi \) corresponds to functional coverage, for which one expects to find executable test cases, whereas \( \pi \models \neg \varphi \) corresponds to requirement violations, for which executions will exist only if the system fails to meet those specifications. Additionally, \( \varphi \) characterizes all the adequate test cases for \( a \) in \( \varphi \), independently of the system. It does not say whether such a test case exists within the system’s valid executions or not (see Figure 7 and the discussion at the end of the previous section).

The definition for \( \varphi \) applies to both finite and infinite paths. If the system at stake features only finite executions, then \( \varphi \) is sufficient in itself. If the system has infinite executions (and finite tests are required), or executions whose length is beyond that of the test cases one is willing to consider, then \( \varphi \) has to be refined with a notion of test prefix, based on weak and strong semantic variants [9], along the lines of [21]. We leave this issue open for future investigation.

Our restriction to linear formulae may seem quite constraining. The situation, however, is not as bad as it seems, and is similar to masking vs. unique-couple MC/DC. Moreover, the issue of linearity (i.e., multiple occurrences of the same atom) is essentially the same issue occurring with strongly coupled atoms (see above the example with \( x \leq 0 \) and \( x > 0 \)). Solutions may be found for special cases (and we leave these for future work), but a general approach seems problematic.

On a different level, our work presents some similarities with the notion of vacuous formulae [16, 1]. A formula \( \varphi \) passes vacuously in a given model \( M \) if there exists a subformula \( \varphi' \) of \( \varphi \) such that the truth value of \( \varphi' \) in \( M \) is the same when \( \varphi' \) is replaced by \( F \). Thus, in order to have an adequate test case for an atom in a formula, the formula must not be vacuous for that atom. However, the definition of vacuity involves explicitly a model, while in our case the trap formulae we derive are independent of the model. We leave for future work a detailed investigation of the relationships between trap formulae and vacuity detection.

In parallel with the verification of models for model checkers, we are currently in the process of applying our metric to the verification of planning domains. In particular, we are developing an application to generate test cases automatically from flight rules for PDDL domains. Preliminary results are reported in [2] and our aim for the future is to deliver a testing platform that integrates our formal approach with the design and development environments currently in use.

**Acknowledgments**

We are grateful to Dimitra Giannakopoulou and Corina Păsăreanu for their comments on a previous version of this work and to the anonymous reviewers for their valuable input.

6. REFERENCES

- M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. The Maude system. In P. Narendran and M. Rusinowitch, editors, Rewriting Techniques and Applications, 10th


