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, 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:

*This work was partially funded by RIACS at NASA Ames and by EPSRC under grant EP/D077273/1 for project UbiVal

ISSTA'09, July 19–23, 2009, Chicago, Illinois, USA.

1. Define in a formal way what it means for an execution path π to be an adequate test case for a formula φ and an atom a appearing in the formula. We will use the notation $\mathsf{FLIP}(\varphi, a)$ to denote the set of execution paths that are adequate tests for a in φ (the meaning of FLIP will become clear in the following sections).

2. Given a formula φ and an atom a, provide a procedure to derive a new formula $[\varphi]_a$ (called a *trap formula*) such that

$$\pi \models [\varphi]_a \iff \pi \in \mathsf{FLIP}(\varphi, a)$$

i.e., the test cases for a formula φ being true in a path π because of atom a are all (and only) the paths π 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 definitions. 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 φ, interpreted over (finite or infinite) paths π, built from a set of states S. By a slight abuse of notation, we equate logic formulae to their semantic validity set and consider that

$$\varphi(\pi) \quad \equiv \quad \pi \models \varphi \quad \equiv \quad \pi \in \varphi$$

Given a finite path $\pi = s_1 \to s_2 \to \cdots \to s_n$ in S^* and $1 \le i \le j \le |\pi|$, we define

$$\begin{aligned} \pi(i) &= s_i \\ \pi(i:j) &= s_i \to \cdots \to s_j \\ \pi(i:) &= s_i \to \cdots \to s_{|\pi|} \end{aligned}$$

By extension, for infinite paths $\pi = s_1 \to s_2 \to \cdots$ in S^ω, $|\pi| = \omega$ and these notations still apply, with $\pi(i:) = s_i \to \cdots$.

2.1 Projections and Variants

Let $AC(\varphi)$ be the set of atomic conditions in a formula φ, 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 π. 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 π (also called the *projection* of π over X).

Definition 1. Given $X \subseteq AC(\varphi)$, a path π' is an X-*variant* of a path of π, denoted $\pi \overset{X}{\leftrightarrow} \pi'$, iff

$$\pi(AC(\varphi) - X) = \pi'(AC(\varphi) - X)$$

In what follows, we will only consider variants with respect to a single condition a, which we will denote $\pi \overset{a}{\leftrightarrow} \pi'$. Obviously, $\overset{a}{\leftrightarrow}$ is an equivalence relation over paths, so each path π induces an equivalence class $[\pi]_a = \{\pi' \mid \pi' \overset{a}{\leftrightarrow} \pi\}$[1]. By construction, if $\pi \overset{a}{\leftrightarrow} \pi'$, then $|\pi| = |\pi'|$ and for any i, j we have that $\pi(i:j) \overset{a}{\leftrightarrow} \pi'(i:j)$.

2.2 Linearity

This section discusses the distinction between atomic *conditions* $a_1, \ldots, a_n$ 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 condition.

Definition 2. A formula φ is *linear* in a condition a iff a occurs only once in φ. It is *constant* in a if a does not occur in φ.

For instance, $\mathsf{F}(a \wedge b) \wedge \mathsf{F}(\neg a \wedge c)$ 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 φ. φ is linear or constant in a iff $count(a, \varphi)$ is 1 or 0. The *linearization* of φ, denoted $lin(\varphi)$ is obtained by replacing every occurrence of any condition a in φ by a distinct variant a^i, where $1 \le i \le count(a, \varphi)$. By construction, $lin(\varphi)$ is linear in all its conditions. We define $unlin(lin(\varphi)) = \varphi$; in particular $unlin(a^i) = a$.

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 π, π' such that $|\pi| = |\pi'|$,

$$\pi \sqsubseteq_a \pi' \quad \iff \quad \pi \overset{a}{\leftrightarrow} \pi' \wedge \forall i \le |\pi| \cdot \pi(i)(a) \Rightarrow \pi'(i)(a)$$

where $\pi(i)(a)$ denotes the value of a in the i-th state of π. Intuitively, this means that a is true "more often" over π' than over π, 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 $\wedge_a, \vee_a, \neg_a$ defined point-wise, e.g. $\pi' \wedge_a \pi''$ is a new path π where $\pi(i)(a) = \pi'(i)(a) \wedge \pi''(i)(a)$. Top and bottom elements of this lattice are $\pi[a := \mathbf{T}]$ and $\pi[a := \mathbf{F}]$, where $\pi[a := v]$ means the a-variant of π where $\pi(i)(a) = v$ for all indices i.

Definition 3. Given a condition a, a temporal formula φ 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 $\pi' \models \varphi$. φ is *monotonic* in a iff it is either covariant or contravariant in a.

All usual logic operators preserve monotonicity, in the following 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 θ is *covariant* (resp. *contravariant*) in a iff for any φ that is covariant in a, $\theta(\varphi)$ is covariant (resp. contravariant) in a, and vice-versa for φ contravariant in a. θ is *monotonic* in a iff it is either covariant or contravariant in a.

A composition $\theta' \circ \theta$ is monotonic if θ and θ' 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 φ', then $\bullet \wedge \varphi'$ and $\bullet \vee \varphi'$ are covariant and $\neg$ is contravariant in a. Note however that $\oplus$ and $\iff$ are not monotonic. All LTL operators ($\mathsf{X}, \mathsf{U}, \mathsf{R}, \mathsf{F}, \mathsf{G}$) 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) $\pi \models \varphi_a \,\mathsf{U}\, \varphi'$ (or any other temporal operator) depends only on $\pi(i:) \models \varphi_a$ and $\pi(j:) \models \varphi'$ on suffixes of π, and $\pi \sqsubseteq_a \pi'$ implies $\pi(i:) \sqsubseteq_a \pi'(i:)$.[2] As a result, if a formula φ (built from these operators) is linear in a then it is monotonic in a.[3]

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 || b) { ... }` contains the decision $c \equiv (a \vee b)$ with conditions a and b. MC/DC is achieved if this decision is exercised with the following three valuations:[4]

a	b	$a \vee b$
$\top$	$\bot$	$\top$
$\bot$	$\top$	$\top$
$\bot$	$\bot$	$\bot$

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 $a = \top$, then a affect φ *positively*, and if $a = \bot$, then a affect φ *negatively.*

[2]More generally, $\pi \sqsubseteq_a \pi'$ implies $\pi(i:j) \sqsubseteq_a \pi'(i:j)$, so this would still be true e.g. for past-tense LTL operators.
[3]More generally, one expects that if φ is covariant in all occurrences of a (i.e. $lin(\varphi)$ is covariant in all a^i) then φ is covariant in a, and similarly for contra-variance.
[4]We use $\top$ and $\bot$ to denote Boolean true and false.

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 $(a \wedge b) \vee (\neg a \wedge c)$).

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 a is the *unique first cause* (UFC) for φ along a path π if, in the first state along π in which φ 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 φ and condition a in φ, a trap formula $\mathrm{ufc}(\varphi, a)$ can be derived such that $\mathrm{ufc}(\varphi, a)$ holds in all suitable execution paths in which a is the first cause for φ. We report here only a few derivation rules that will be used below and refer to [21] for further details (φ_a denotes a formula in which a occurs):

$$\begin{aligned}
\mathrm{ufc}(\varphi, a) &= \mathbf{F} \qquad \text{where } a \text{ does not occur in } \varphi \\
\mathrm{ufc}(a, a) &= a \\
\mathrm{ufc}(\varphi_a \vee \psi) &= \mathrm{ufc}(\varphi_a, a) \wedge \neg\psi \\
\mathrm{ufc}(\mathsf{F}\,\varphi_a, a) &= \neg\varphi_a \,\mathsf{U}\, \mathrm{ufc}(\varphi_a, a)
\end{aligned}$$

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 $\mathrm{ufc}(\varphi, a)$ corresponds indeed to the fact that "φ 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 $\varphi = \mathsf{F}\,(a \vee b)$. The derivation rules presented above give

$$\mathrm{ufc}(\varphi_a, a) = (\neg a \wedge \neg b) \,\mathsf{U}\, (a \wedge \neg b)$$

The trace $\pi = \{\} \to \{a\} \to \{b\}$ does satisfy $\mathrm{ufc}(\varphi, a)$. 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 $\mathsf{F}\,(a \vee b)$ false along this trace (cf. the criteria

appearing in MC/DC). In essence, using the definition of a-variant defined above, there is no a-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 $\mathrm{ufc}(\varphi, a)$ is satisfied along all valid test cases, but it is also satisfied on execution paths that are not adequate test cases, i.e., $\mathrm{ufc}(\varphi, a)$ is "too generous". Equivalently, as it will be described below, $\mathrm{ufc}(\varphi, a)$ 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 iff $\pi \models \varphi$ and there exists an a-variant π' of π such that $\pi' \not\models \varphi$. We denote with $\mathsf{FLIP}(\varphi, a)$ the set of all such paths.

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

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

$$
\begin{array}{rcl}
[\varphi']_a & = & \mathbf{F} \qquad \text{where } a \text{ does not occur in } \varphi' \\
[a]_a & = & a \\
[\neg a]_a & = & \neg a \\
[\varphi_a \wedge \varphi']_a & = & [\varphi_a]_a \wedge \varphi' \\
[\varphi_a \vee \varphi']_a & = & [\varphi_a]_a \wedge \neg\varphi' \\
[\mathsf{X}\,\varphi_a]_a & = & \mathsf{X}\,[\varphi_a]_a \\
[\varphi'\,\mathsf{U}\,\varphi_a]_a & = & (\varphi'\,\mathsf{U}\,\varphi_a) \wedge (\neg\varphi'\,\mathsf{R}\,(\varphi_a \Rightarrow [\varphi_a]_a)) \\
[\varphi_a\,\mathsf{U}\,\varphi']_a & = & (\varphi_a\,\mathsf{U}\,\varphi') \wedge (\neg\varphi'\,\mathsf{U}\,([\varphi_a]_a \wedge \neg\varphi')) \\
[\varphi_a\,\mathsf{R}\,\varphi']_a & = & (\varphi_a\,\mathsf{R}\,\varphi') \wedge ((\varphi_a \Rightarrow [\varphi_a]_a)\,\mathsf{U}\,\neg\varphi') \\
[\varphi'\,\mathsf{R}\,\varphi_a]_a & = & (\varphi'\,\mathsf{R}\,\varphi_a) \wedge (\neg\varphi'\,\mathsf{U}\,[\varphi_a]_a)
\end{array}
$$

(where R is the standard release operator).

Other cases are obtained by syntactic derivation:

$$
\begin{array}{rcl}
[\mathsf{F}\,\varphi_a]_a & = & \mathsf{F}\,\varphi_a \wedge \mathsf{G}\,(\varphi_a \Rightarrow [\varphi_a]_a) \\
[\mathsf{G}\,\varphi_a]_a & = & \mathsf{G}\,\varphi_a \wedge \mathsf{F}\,[\varphi_a]_a \\
[\varphi'\,\mathsf{W}\,\varphi_a]_a & = & (\varphi'\,\mathsf{W}\,\varphi_a) \wedge ((\varphi_a \Rightarrow [\varphi_a]_a) \\
 & & \quad \mathsf{U}\,(\neg\varphi' \wedge (\varphi_a \Rightarrow [\varphi_a]_a))) \\
[\varphi_a\,\mathsf{W}\,\varphi']_a & = & (\varphi_a\,\mathsf{W}\,\varphi') \wedge (\neg\varphi'\,\mathsf{U}\,(\neg\varphi' \wedge [\varphi_a]_a))
\end{array}
$$

These derivations for fixed point modalities are all of the form $[\varphi]_a = \varphi \wedge \varphi''$, where the recursive step occurs only in φ''. They can be rewritten into equivalent forms $[\varphi]_a = \varphi_1\,\mathsf{U}\,([\varphi_a]_a \wedge \varphi_2)$:

$$
\begin{array}{rcl}
[\varphi'\,\mathsf{U}\,\varphi_a]_a & = & (\varphi' \wedge \neg\varphi_a)\,\mathsf{U}\,([\varphi_a]_a \wedge (\neg\varphi'\,\mathsf{R}\,(\varphi_a \Rightarrow [\varphi_a]_a)) \\
[\varphi_a\,\mathsf{U}\,\varphi']_a & = & (\varphi_a \wedge \neg\varphi')\,\mathsf{U}\,([\varphi_a]_a \wedge \neg\varphi' \wedge (\varphi_a\,\mathsf{U}\,\varphi')) \\
[\varphi_a\,\mathsf{R}\,\varphi']_a & = & (\neg\varphi_a \wedge \varphi')\,\mathsf{U}\,([\varphi_a]_a \wedge \varphi' \wedge \\
 & & \quad (\varphi_a \Rightarrow [\varphi_a]_a)\,\mathsf{U}\,\neg\varphi')) \\
[\varphi'\,\mathsf{R}\,\varphi_a]_a & = & (\neg\varphi' \wedge \varphi_a)\,\mathsf{U}\,([\varphi_a]_a \wedge (\varphi'\,\mathsf{R}\,\varphi_a)) \\
[\mathsf{F}\,\varphi_a]_a & = & \neg\varphi_a\,\mathsf{U}\,([\varphi_a]_a \wedge \mathsf{G}\,(\varphi_a \Rightarrow [\varphi_a]_a)) \\
[\mathsf{G}\,\varphi_a]_a & = & \varphi_a\,\mathsf{U}\,([\varphi_a]_a \wedge \mathsf{G}\,\varphi_a)
\end{array}
$$

Proof of correctness. We now show that $\pi \models [\varphi]_a$ if and only if $\pi \in \mathsf{FLIP}(\varphi, a)$. We prove completeness (only if) and soundness (if) separately.

We first need the following simple lemma:

THEOREM 1 (EXTREMAL a-VARIANTS). *If φ is covariant (resp. contravariant) in a and $\pi \in [\varphi]_a$ then $\pi[a := \mathbf{F}] \not\models \varphi$ (resp. $\pi[a := \mathbf{T}] \not\models \varphi$).*

PROOF. We prove the covariant case; the contravariant case follows by duality. Since $\pi \in [\varphi]_a$, there is a $\pi' \overset{a}{\leftrightarrow} \pi$ such that $\pi' \not\models \varphi$. By contra-position of covariance, if $\pi'' \sqsubseteq_a \pi'$ and $\pi' \not\models \varphi$ then $\pi'' \not\models \varphi$. But $\pi[a := \mathbf{F}]$ is minimal in $[\pi]_a$, therefore $\pi[a := \mathbf{F}] \sqsubseteq_a \pi'$ and $\pi[a := \mathbf{F}] \not\models \varphi$. □

THEOREM 2 (SOUNDNESS OF $[\varphi]_a$). *Let φ be a formula in negation-normal form, linear and monotonic in a. If $\pi \models [\varphi]_a$, then $\pi \models \varphi$ and there exists $\pi' \overset{a}{\leftrightarrow} \pi$ such that $\pi \not\models \varphi$ (i.e., $\pi \in \mathsf{FLIP}(\varphi, a)$).*

PROOF. Let $\pi \models [\varphi]_a$. We have to show that $\pi \models \varphi$ and build π' such that $\pi' \overset{a}{\leftrightarrow} \pi$ and $\pi' \not\models \varphi$. By induction, we can assume that for any sub-formula φ_1 of φ, if $\pi_1 \models [\varphi_1]_a$, then $\pi_1 \models \varphi_1$ and there is $\pi_1' \overset{a}{\leftrightarrow} \pi_1$ such that $\pi_1' \not\models \varphi_1$.

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

- $[\varphi]_a = \mathbf{F}$, where a does not occur in φ:

  Trivially, there is no such π.

- $[a]_a = a$:

  Obviously, $\pi \models \varphi$, which means $\pi(1)(a) = \mathbf{T}$ and $\pi' = \pi[a := \mathbf{F}] \not\models \varphi$.

- $[\neg a]_a = \neg a$:

  Dual of the previous case: $\pi(1)(a) = \mathbf{F}$ and $\pi' = \pi[a := \mathbf{T}] \not\models \varphi$.

- $[\varphi_a \wedge \varphi']_a = [\varphi_a]_a \wedge \varphi'$:

  Since $\pi \models [\varphi_a]_a$, $\pi \models \varphi_a$ and there is $\pi' \overset{a}{\leftrightarrow} \pi$ such that $\pi' \not\models \varphi_a$ (and a does not occur in φ'). Hence $\pi \models \varphi_a \wedge \varphi'$ and $\pi' \not\models \varphi_a \wedge \varphi'$.

- $[\varphi_a \vee \varphi']_a = [\varphi_a]_a \wedge \neg\varphi'$:

  Similar to $\varphi_a \wedge \varphi'$, but in this case $\pi \not\models \varphi'$ so $\pi' \not\models \varphi'$.

- $[\mathsf{X}\,\varphi_a]_a = \mathsf{X}\,[\varphi_a]_a$:

  We have $\pi(2:) \models [\varphi_a]_a$ so (i) $\pi(2:) \models \varphi_a$ and thus $\pi \models \varphi$ and (ii) there is $\pi_2' \overset{a}{\leftrightarrow} \pi(2:)$ such that $\pi_2' \not\models \varphi_a$. Build $\pi'(1) = \pi(1)$ and $\pi'(2:) = \pi_2'$, thus $\pi' \not\models \varphi$.

- $[\varphi'\,\mathsf{U}\,\varphi_a]_a = (\varphi'\,\mathsf{U}\,\varphi_a) \wedge (\neg\varphi'\,\mathsf{R}\,(\varphi_a \Rightarrow [\varphi_a]_a))$:

  Obviously $\pi \models \varphi$. Let n the first index such that $\pi(n:) \models \neg\varphi'$ (n may be infinite). Consider all $j \leq n$ such that $\pi(j:) \models \varphi_a$. Because $\pi \models (\neg\varphi'\,\mathsf{R}\,(\varphi_a \Rightarrow [\varphi_a]_a))$, we have $\pi(j:) \models [\varphi_a]_a$. Let $\pi' = \pi[a := \mathbf{F}]$. By monotonicity of φ_a (theorem 1), $\pi'(j:) \not\models \varphi_a$ for all $j \leq n$. Therefore there is no $j \leq n$ such that $\pi'(j:) \models \varphi_a$, so $\pi' \not\models \varphi$.

	s_1	$\cdots$	s_{m-1}	s_m	$\cdots$	s_j	$\cdots$	s_{n-1}	s_n	$\cdots$
π, π' :	φ'	φ'	φ'	φ'	φ'	φ'	φ'	φ'	$\boxed{\neg\varphi'}$	$\cdots$
π :	$\neg\varphi_a$	$\neg\varphi_a$	$\neg\varphi_a$	$\boxed{\varphi_a}$	$\cdots$	φ_a	$\cdots$	$\cdots$	$\cdots$	$\cdots$
π' :	$\neg\varphi_a$	$\neg\varphi_a$	$\neg\varphi_a$	$\neg\varphi_a$	$\cdots$	$\neg\varphi_a$	$\cdots$	$\cdots$	$\cdots$	$\cdots$

Figure 1: Execution paths for $\varphi' \,\mathsf{U}\, \varphi_a$.

- $[\varphi_a \,\mathsf{R}\, \varphi']_a = (\varphi' \,\mathsf{U}\, (\varphi' \wedge \varphi_a)) \wedge ((\varphi_a \Rightarrow [\varphi_a]_a) \,\mathsf{U}\, \neg\varphi')$:
  Obviously $\pi \models \varphi$. Since $\pi \models ((\varphi_a \Rightarrow [\varphi_a]_a)\mathsf{U}\neg\varphi')\mathsf{U}\neg\varphi'$, there is a minimal finite n such that $\pi(n:) \models \neg\varphi'$. The proof follows similarly as for $\varphi' \,\mathsf{U}\, \varphi_a$, with the difference that the finite n guarantees that $\varphi_a \,\mathsf{R}\, \varphi'$ is indeed eventually falsified along $\pi[a := \mathbf{F}]$.
- $[\varphi_a \,\mathsf{U}\, \varphi']_a = (\varphi_a \,\mathsf{U}\, \varphi') \wedge (\neg\varphi' \,\mathsf{U}\, ([\varphi_a]_a \wedge \neg\varphi'))$:
  Obviously $\pi \models \varphi$. Let n the first index such that $\pi(n:) \models \varphi'$ (n must be finite). Because $\pi \models \neg\varphi' \,\mathsf{U}\, ([\varphi_a]_a \wedge \neg\varphi')$, there must be $k < n$ such that $\pi(k:) \models [\varphi_a]_a$. Let $\pi' = \pi[a := \mathbf{F}]$. By monotonicity of φ_a, we have $\varphi'(k:) \not\models \varphi_a$, hence $\pi' \not\models \varphi_a \,\mathsf{U}\, \varphi'$.
- $[\varphi' \,\mathsf{R}\, \varphi_a]_a = (\varphi' \,\mathsf{R}\, \varphi_a) \wedge (\neg\varphi' \,\mathsf{U}\, [\varphi_a]_a)$:
  The proof is similar to that for $\varphi_a \,\mathsf{U}\, \varphi'$, except that n may be infinite and k may be equal to n.

□

THEOREM 3 (COMPLETENESS OF $[\varphi]_a$). *Let φ be a formula in negation-normal form, linear and monotonic in a. If $\pi \models \varphi$ and there exists $\pi' \overset{a}{\leftrightarrow} \pi$ such that $\pi \not\models \varphi$ (i.e., if $\pi \in \mathsf{FLIP}(\varphi, a)$), then $\pi \models [\varphi]_a$.*

PROOF. Let $\pi' \overset{a}{\leftrightarrow} \pi$ such that $\pi \models \varphi$ and $\pi' \not\models \varphi$. We have to show that $\pi \models [\varphi]_a$. By induction, we can assume that for any sub-formula φ_1 of φ, if there is $\pi_1' \overset{a}{\leftrightarrow} \pi_1$ such that $\pi_1 \models \varphi_1$ and $\pi_1' \not\models \varphi_1$, then $\pi_1 \models [\varphi_1]_a$.

- $[\varphi']_a = \mathbf{F}$, where a does not occur in φ':
  If a does not occur in φ' and $\pi \models \varphi'$, then for all $\pi' \overset{a}{\leftrightarrow} \pi$ we also have $\pi' \models \varphi'$.
- $[a]_a = a$:
  Obviously, $\pi \models a$.
- $[\neg a]_a = \neg a$:
  Obviously, $\pi \models \neg a$.
- $[\varphi_a \wedge \varphi']_a = [\varphi_a]_a \wedge \varphi'$:
  We have $\pi \models \varphi_a$ and $\pi \models \varphi'$ so $\pi' \models \varphi'$ because φ' does not depend on a. However $\pi' \not\models \varphi_a \wedge \varphi'$ so we must have $\pi' \not\models \varphi_a$. Hence $\pi \models [\varphi_a]_a \wedge \varphi'$.
- $[\varphi_a \vee \varphi']_a = [\varphi_a]_a \wedge \neg\varphi'$:
  Similar to $\varphi_a \wedge \varphi'$ except $\pi \not\models \varphi'$ so $\pi' \not\models \varphi'$.
- $[\mathsf{X}\, \varphi_a]_a = \mathsf{X}\, [\varphi_a]_a$:
  We have $\pi' \not\models \mathsf{X}\varphi_a$. In general, $\neg\mathsf{X}\varphi_a = \mathsf{X}\neg\varphi_a \vee \neg\mathsf{X}\,\mathbf{T}$. However, $\pi' \overset{a}{\leftrightarrow} \pi$ so π and π' must have the same length: since $\pi \models \mathsf{X}\,\mathbf{T}$, we also have $\pi' \models \mathsf{X}\,\mathbf{T}$ and therefore $\pi' \models \mathsf{X}\,\neg\varphi_a$. Hence $\pi(2:) \models \varphi_a$ and $\pi'(2:) \models \neg\varphi_a$, so $\pi(2:) \models [\varphi_a]_a$.
- $[\varphi' \,\mathsf{U}\, \varphi_a]_a = (\varphi' \,\mathsf{U}\, \varphi_a) \wedge (\neg\varphi' \,\mathsf{R}\, (\varphi_a \Rightarrow [\varphi_a]_a))$:
  We readily have $\pi \models \varphi' \,\mathsf{U}\, \varphi_a$. Let m, n the first indices such that $\pi(m:) \models \varphi_a$ and $\pi(n:) \models \neg\varphi'$. Because $\pi \models \varphi' \,\mathsf{U}\, \varphi_a$ we have that m is finite and $m \leq n$ (n may be infinite).
  For all $i < n$ we have $\pi'(i:) \models \varphi'$, and $\pi'(n:) \models \neg\varphi'$, since a does not occur in φ'. Let $m \leq j \leq n$ such that $\pi(j:) \models \varphi_a$. We have that $\pi'(j:) \models \neg\varphi_a$, otherwise $\pi' \models \varphi$. Hence $\pi(j:) \models [\varphi_a]_a$ by inductive hypothesis, and $\pi \models \neg\varphi' \,\mathsf{R}\, (\varphi_a \Rightarrow [\varphi_a]_a)$ (see Figure 1 for a visual representation).
- $[\varphi_a \,\mathsf{R}\, \varphi']_a = (\varphi' \,\mathsf{U}\, (\varphi' \wedge \varphi_a)) \wedge ((\varphi_a \Rightarrow [\varphi_a]_a) \,\mathsf{U}\, \neg\varphi')$:
  This case is similar to $\varphi' \,\mathsf{U}\, \varphi_a$, except that n must be finite, for otherwise $\pi' \models \mathsf{G}\, \varphi'$ and thus $\pi' \models \varphi$. Also, in this case $m < n$ and it is enough to consider all $j < n$, since $\pi \not\models \varphi_a \,\mathsf{R}\, \varphi'$ if $m = n$.
- $[\varphi_a \,\mathsf{U}\, \varphi']_a = (\varphi_a \,\mathsf{U}\, \varphi') \wedge (\neg\varphi' \,\mathsf{U}\, ([\varphi_a]_a \wedge \neg\varphi'))$:
  We readily have $\pi \models \varphi_a \,\mathsf{U}\, \varphi'$. Let n the first (finite) index such that $\pi(n:) \models \varphi'$ (and similarly for φ' because it is independent on a). For all $i < n$ we have $\pi(i:) \models \varphi_a$. Conversely, there must be $j < n$ such that $\pi'(j:) \not\models \varphi_a$, for otherwise $\pi' \models \varphi_a \,\mathsf{U}\, \varphi'$. Hence $\pi(j:) \models [\varphi_a]_a$ and $\pi \models \neg\varphi' \,\mathsf{U}\, ([\varphi_a]_a \wedge \neg\varphi')$.
- $[\varphi' \,\mathsf{R}\, \varphi_a]_a = (\varphi' \,\mathsf{R}\, \varphi_a) \wedge (\neg\varphi' \,\mathsf{U}\, [\varphi_a]_a)$:
  This case is similar to $\varphi_a \,\mathsf{U}\, \varphi'$, except that n may be infinite and all $i \leq n$ have to be considered (possibly infinitely many). On the other hand, j must be finite, which ensures $\neg\varphi' \,\mathsf{U}\, [\varphi_a]_a$.

□

Theorems 2 and 3 prove that our syntactic derivation of trap formulae is sound and complete with respect to our definition of adequate test cases (Definition 5). As a simple example, consider again the formula $\varphi = \mathsf{F}\, (a \vee b)$ introduced in Section 3.2 to illustrate the issues with the notion of unique first cause of [21]. Following our derivation rules, the trap formula for atom a is given by:

$$[\mathsf{F}\, (a \vee b)]_a = \mathsf{F}\, (a \vee b) \wedge \mathsf{G}\, ((a \vee b) \Rightarrow [a \vee b]_a)$$

Given that $[a \vee b]_a = (a \wedge \neg b)$ and that $(a \vee b) \Rightarrow (a \wedge \neg b)$ is equivalent to $\neg b$, we have that

$$[\mathsf{F}\, (a \vee b)]_a = \mathsf{F}\, (a) \wedge (\mathsf{G}\, (\neg b))$$

Intuitively, this formula says that an adequate test case for $\mathsf{F}\, (a \vee b)$ *because of atom a* is a path where eventually $(a \vee b)$ holds, but nowhere b holds. Indeed, if b were true anywhere in the path, then it would not be possible to flip the value of the original formula because of a. Notice how

```
op [_]_ : Formula Atom -> Formula [prec 10] .
vars C C' C'' : Atom .
vars X Y Z : Formula .
eq [C]C = C .
eq [C']C = False [otherwise] .
eq [~ C]C = ~ C .
eq [~ C']C = False [otherwise] .
ceq [X /\ Y]C = ([X]C /\ Y)
    if C in X .
ceq [X \/ Y]C = ([X]C /\ ~ Y)
    if C in X .
eq [O X]C = O [X]C .
ceq [X U Y]C = (X U Y) /\ (~ Y U ([X]C /\ ~ Y))
    if C in X .
ceq [X U Y]C = (X U Y) /\ (~ X R (Y -> [Y]C))
    if C in Y .
ceq [X R Y]C = (X R Y) /\ ((X -> [X]C) U ~ Y)
    if C in X .
ceq [X R Y]C = (X R Y) /\ (~ X U [Y]C)
    if C in Y .
ceq [X]C = False
    if not (C in X) .
```

Figure 2: Maude definition of FLIP.

our characterisation does not allow for the problematic path $\pi = \{\} \rightarrow \{a\} \rightarrow \{b\}$ 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
```

which takes a formula φ and an atom, and returns a new formula ψ representing the trap formula for the atom in the original formula φ. An extract from the module definition is reported in Figure 2. The module, together with all the examples reported below, is available from `http:/www.cs.ucl.ac.uk/staff/f.raimondi/flip/`.

As an example, consider the following requirement from [8]:

> **REQUIREMENT**: All messages incoming from the POP server should be marked as unread.
> **REFINEMENT**: `MarkasUnread` will occur after `PlacedinMailboxes`[5].

This is encoded by the following LTL formula:

$$\mathsf{G}\,(\texttt{PlacedinMailboxes} \Rightarrow \mathsf{F}\,(\texttt{MarkasUnread}))$$

Suppose we want to derive a test in which the formula is true because of `MarkasUnread`. The output of Maude is reported in Figure 3 (where `PM` corresponds to `PlacedinMailboxes` and `MU` to `MarkasUnread`).

The resulting formula inf Figure 3 can be simplified to

$$\mathsf{F}\,(\neg\texttt{PlacedinMailboxes} \wedge \mathsf{G}\,(\neg\texttt{MarkasUnread})) \wedge \mathsf{G}\,(\texttt{PlacedinMailboxes} \Rightarrow \mathsf{F}\,(\texttt{MarkasUnread}))$$

[5]See `http://patterns.projects.cis.ksu.edu/documentation/specifications/ALL.raw`.

```
                    \||||||||||||||||||/
                  --- Welcome to Maude ---
                    /||||||||||||||||||\
          Maude 2.4 built: Nov  6 2008 16:49:57
          Copyright 1997-2008 SRI International
                Mon Jan  5 10:13:41 2009
Maude> load model-checker.maude
Maude> load ufc.maude
Maude> Maude> red [ []( 'PM -> <> ( 'MU ) ) ] 'PM .
result Formula: (True U (~ 'PM /\ (False R ~ 'MU))) /\
      (False R (~ 'PM \/ (True U 'MU)))
```

Figure 3: Maude screen shot

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:

$$\varphi_{ME} = \mathsf{G}\,((\texttt{p1e} \vee \texttt{p2e}) \Rightarrow \mathsf{F}\,(\texttt{p1c} \vee \texttt{p2c}))$$

where `p1e` 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 φ_{ME} holds. We can use our Maude module to derive the trap formula encoding an adequate test case for `p1c` (i.e., for process 1 in the critical state, see Figure 5). The trap formula can be simplified to:

$$\mathsf{F}\,(\mathsf{F}\,(\texttt{p1c} \vee \texttt{p2c}) \wedge \mathsf{G}\,(\neg\texttt{p2c}) \wedge (\texttt{p1e} \wedge \texttt{p2e})) \wedge \mathsf{G}\,((\texttt{p1e} \vee \texttt{p2e}) \Rightarrow \mathsf{F}\,(\texttt{p1c} \vee \texttt{p2c}))$$

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 truth 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 `p1c` can flip the value of the formula. Indeed, if such an execution exists,

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

MODULE main
 VAR
   semaphore : boolean;
   proc1     : process user(semaphore);
   proc2     : process user(semaphore);
 ASSIGN
   init(semaphore) := 0;

 FAIRNESS
   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 φ_{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 $[\varphi]_a$ encoding a adequate test cases for an atom a in a formula φ. 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 φ is a desired ("positive") property, and $\neg[\varphi]_a$ is false 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 φ 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 φ is a desired ("positive") property, and $\neg[\varphi]_a$ is 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 φ *fails* (i.e., it is not possible to cover a in φ 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 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" test 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 φ is a property that should be false ("negative" property), and $\neg[\varphi]_a$ is true in the model: this is the expected outcome and the test succeeds (case (d) in the Figure).

- The formula φ is a property that should be false ("negative" property) property, and $\neg[\varphi]_a$ is 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 $[\varphi]_a$ in M. Notice that $\neg[\varphi]_a$ 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.

```
                    \||||||||||||||||||/
                  --- Welcome to Maude ---
                    /||||||||||||||||||\
           Maude 2.4 built: Nov  6 2008 16:49:57
           Copyright 1997-2008 SRI International
                 Mon Jan  5 10:13:41 2009
Maude> load model-checker.maude
Maude> load ufc.maude
Maude> red [ [](('p1e /\ 'p2e) -> <> ('p1c \/ 'p2c)) ] 'p1c .
reduce in KSU6 : [[](('p1e /\ 'p2e -> <> ('p1c \/ 'p2c))]'p1c .
rewrites: 299 in 7ms cpu (7ms real) (41447 rewrites/second)
result Formula: (True U 'p1e /\ 'p2e /\ ((True U 'p1c \/ 'p2c)
  /\ (False R ~ 'p2c))) /\ (False R ~ 'p1e \/ ~ 'p2e \/
  (True U 'p1c \/ 'p2c))
```

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

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

```
-- specification !( G ((proc1.state = entering
   & proc2.state = entering) ->  F (proc1.state =
   critical | proc2.state = critical)) &
   F (((proc1.state = entering & proc2.state
   = entering) &  F (proc1.state = critical |
   proc2.state = critical)) &
   G !(proc2.state = critical)))  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-> State: 2.1 <-
  semaphore = 0
  proc1.state = idle
  proc2.state = idle
-> Input: 2.2 <-
[...]
```

Figure 6: NuSMV counterexample for a trap formula.

$F = \mathbf{T}$ and $F = \mathbf{F}$ correspond to alternate paths in the program, that one is a priori equally interested in covering. In FLIP coverage of a requirement φ, 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, $[\varphi]_a$ characterizes all the adequate test cases for a in φ, 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 $[\varphi]_a$ applies to both finite and finite paths. If the system at stake features only finite executions, then $[\varphi]_a$ 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 $[\varphi]_a$ 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-cause 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 φ *passes vacuously* in a given model M if there exists a subformula φ' of φ such that the truth value of φ in M is the same when φ' is replaced by $\mathbf{F}$. 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 vacuosity 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 vacuosity 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

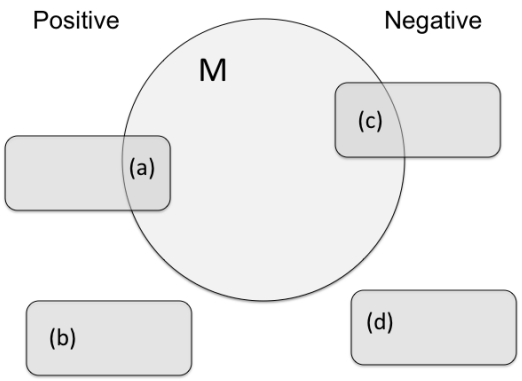

Figure 7: Possible outcomes of the verification of a trap formula with a model checker

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 Pasareanu for their comments on a previous version of this work and to the anonymous reviewers for their valuable input.

6. REFERENCES

[1] R. Armoni, L. Fix, A. Flaisher, O. Grumberg, N. Piterman, A. Tiemeyer, and M. Y. Vardi. Enhanced vacuity detection in linear temporal logic. In *CAV 2003, Boulder, CO, USA*, volume 2725 of *Lecture Notes in Computer Science*, pages 368–380. Springer, 2003.

[2] G. Brat, C. Pecheur, and F. Raimondi. A formal analysis of requirements-based testing and its applications to the verification of spin, nusmv, and pddl domains. Technical report, NASA, March 2009.

[3] J. J. Chilenski. An investigation of three forms of the modified condition decision coverage (MCDC) criterion. Technical report DOT/FAA/AR-01/18DOT/FAA/AR-01/18, Federal Aviation Administration, 2001.

[4] J. J. Chilenski and S. P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, pages 193–200, 1994.

[5] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new symbolic model verifier. In *Proc. of International Conference on Computer-Aided Verification*, 1999.

[6] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. Mit Press, 1999.

[7] 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*

International Conference, RTA'99, Trento, Italy, July 2–4, 1999, volume 1631 of *Lecture Notes in Computer Science*, pages 240–243. Springer-Verlag, 1999.

[8] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property specification patterns for finite-state verification. In M. Ardis, editor, *Proceedings of the 2nd Workshop on Formal Methods in Software Practice (FMSP'98)*, pages 7–15, New York, 1998. ACM Press.

[9] C. Eisner et al. Reasoning with temporal logic on truncated paths. In *Proceedings of CAV '03*, volume 2725 of *Lecture Notes in Computer Science*. Springer Verlag, 2003.

[10] A. Gerevini and D. Long. Plan constraints and preferences in pddl3: The language of the fifth international planning competition. Technical report, Dept. of Electronics and Automation, University of Brescia, August 2005.

[11] K. J. Hayhurst, D. S. Veerhusen, J. Chilenski, and L. K. Riersn. A practical tutorial on modified condition/decision coverage. Technical Report TM-2001-210876, NASA Langley Research Center, 2001.

[12] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), 1997.

[13] H. S. Hong, I. Lee, O. Sokolsky, and H. Ural. A temporal logic based theory of test coverage and generation. In *TACAS '02: Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 327–341, London, UK, 2002.

[14] L. Khatib, N. Muscettola, and K. Havelund. Verification of plan models using UPPAAL. *Lecture Notes in Computer Science*, 1871, 2001.

[15] J. Penix, C. Pecheur, and K. Havelund. Using model checking to validate AI planner domain models. In *Proceedings of the 23rd Annual Software Engineering Workshop*, NASA Goddard, 1998.

[16] M. Purandare and F. Somenzi. Vacuum cleaning ctl formulae. In *CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification*, pages 485–499, London, UK, 2002. Springer-Verlag.

[17] RTCA. *Software Considerations in Airborne Systems and Equipment Certification*, 1992.

[18] S. W. Squyres et al. The Opportunity Rover's Athena Science Investigation at Meridiani Planum, Mars. *Science*, 306:1698–1703, 2004.

[19] S. W. Squyres et al. The Spirit Rover's Athena Science Investigation at Gusev Crater, Mars. *Science*, 305:794–799, 2004.

[20] L. Tan, O. Sokolsky, and I. Lee. Specification-based testing with linear temporal logic. In *Proceedings of the IEEE International Conference on Information Reuse and Integration (IRI04)*. IEEE Society, 2004.

[21] M. W. Whalen, A. Rajan, M. P. E. Heimdahl, and S. P. Miller. Coverage metrics for requirements-based testing. In *ISSTA'06: Proceedings of the 2006 international symposium on software testing and analysis*, pages 25–36, New York, 2006. ACM Press.