Testing Planning Domains (without Model Checkers)

Franco Raimondi[1⋆] and Charles Pecheur[2] and Guillaume Brat[3]

[1] Department of Computer Science, University College London, London, UK
[2] Dept. INGI, Université catholique de Louvain, Louvain-la-neuve, Belgium
[3] RIACS/NASA Ames Research Center, Moffett Field, CA, USA

Abstract. We address the problem of verifying planning domains as used in model-based planning, for example in space missions. We propose a methodology for testing flight rules of planning domains which is self-contained, in the sense that flight rules are verified using a planner and no external tools are required. We review and analyse coverage conditions for requirements-based testing, and we reason in detail on "Unique First Cause" (UFC) coverage for test suites. We characterise flight rules using patterns, encoded using LTL, and we provide UFC coverage for them. We then present a translation of LTL formulae into planning goals, and illustrate our approach on a case study.

1 Introduction

The NASA rovers Spirit and Opportunity [1, 2], exploring the surface of Mars since January 2004, are an example of complex systems, featuring significant autonomous capabilities, that can be built using current software and hardware technologies. Due to the complexity of such systems and in order to avoid economic losses and mission failures, there is a growing interest in tools and methodologies to perform formal verification for this kind of autonomous applications. In this paper we are concerned with the problem of verifying *planning domains*. In the case of the Mars rovers, plans are generated using the Europa 2 planner [3] to satisfy some scientific and positioning goals. Then, the plans are manually checked (against a set of requirements called *flight rules*) before being uploaded to the rover. The generation and verification have to be done before the next Mars daytime cycle. The methodology we propose to verify planning domains would speed up the verification phase and help ensure that flight rules are not violated.

Verification of planning domains has been investigated in the past, for instance in [4, 5]. The solutions proposed by these authors consist in the translation of the planning domain into the input language of some model checker. The main limitation of these approaches is the limited size of the domains that can be translated and the problematic correspondence between languages for planners and languages for model checkers. In this paper we suggest a different approach:

⋆ The author ackwoledge support from the European IST project PLASTIC and from Mission Critical Technologies, NASA Ames Research Center

we propose to translate the problem of verification of planning domains into a *planning problem*. Such an approach has the advantage that no external tools are required, because the actual planner can be used to perform verification. Specifically, we suggest to proceed as summarised in Figure 1: given as input a planning domain and a set of flight rules (this is normally provided as a text document), in step 1 we encode flight rules as LTL specifications. In the second step we derive test cases from the specifications; we employ a revised notion of requirements-based testing, using an approach similar to [6–8]. In the third step we convert the test cases generated into planning goals. The actual tests are performed by "enriching" the original planning model with the new set of goals, using the planner for which the domain was designed.

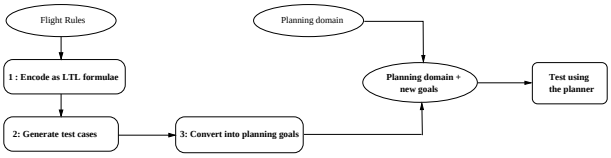

Fig. 1. From flight rules to planning goals (overview).

The rest of the paper is organised as follows. In Section 2 we review the notion of coverage for requirements-based testing. We present the planning problem in Section 3. A motivational example is introduced in Section 4. We show how flight rules can be encoded as temporal formulae in Section 5, and how these can be translated into planning goals in Section 6, using the example provided. We conclude in Section 7.

2 Testing and requirements-based testing

Various metrics exist to quantify the coverage of test suites [9], particularly for *structural* testing. Coverage metrics for *functional* testing can be defined as well when functional requirements are provided formally, for instance using temporal formulae. Previous work in this direction include [8, 7, 6]. In this section we briefly review MC/DC (structural) coverage, and then we reason on a metric for functional testing by extending some concepts from [6].

2.1 MC/DC Coverage

In the scope of this paper, we use a metric inspired by the popular *Modified Condition/Decision Coverage* (MC/DC) [10]. In particular, MC/DC coverage is required for the most critical categories of avionic software [11] and its extensions can be employed in specification-based testing (see below). MC/DC coverage 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]
(1) $a = \top, b = \bot, c = \top$; (2) $a = \bot, b = \top, c = \top$; (3) $a = \bot, b = \bot, c = \bot$;
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.

The MC/DC requirements for each condition can be captured by a pair of Boolean formulae, called *trap formulae*, capturing those valuations in which the condition is shown to positively and negatively affect the decision in which it occurs (also called the positive and the negative test cases). Coverage is achieved by building test cases that exercise the condition in states which satisfy each trap formula. In the example above, the trap formulae for condition a in c are $a \wedge \neg b$ and $\neg a \wedge \neg b$.

2.2 UFC Coverage

As mentioned above, if functional specifications are expressed in a formal framework, then functional testing can be measured in terms of coverage criteria similar to structural testing, but applied to the specifications rather than the implementation. This is the idea behind *requirements-based testing*, as investigated in [6] and in [7]. In particular, [6] defines the notion of *Unique First Cause coverage* (UFC), which extends ideas from MC/DC to requirements-based testing.

UFC coverage is defined with respect to functional properties that executions of a system must satisfy, and to the atomic conditions occurring in these properties. As illustrated in Section 5, these properties are often expressed by means of temporal formulae, for instance using the logic LTL (we refer to [12] for more details). A test suite achieves UFC coverage of a set of requirements expressed as temporal formulae, if: (1) Every basic condition in any formula has taken on all possible outcomes at least once. (2) Each basic condition has been shown to affect the formula's outcome as the *unique first cause*.

Following [6], 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 can be formalised as follows. Let π be a a path and X a set of atomic conditions; we denote by $\pi(X)$ the sequence of truth values of the atomic conditions in X along π (also called the *projection* of π over X).

Definition 1 (a-variant). *Let $AC(\varphi)$ be the set of occurrences of atomic conditions in a linear temporal formula φ.*[5] *Given $a \in AC(\varphi)$ and a path π, an* a-variant *of π w.r.t. φ is a path π' such that $\pi(AC(\varphi) - \{a\}) = \pi'(AC(\varphi) - \{a\})$.*

[4] We use $\top$ and $\bot$ to denote Boolean true and false.

[5] Note that different *occurrences* of the same *condition* a are considered distinct. This poses technical difficulties with multiple occurrences. This is a known issue in the MC/DC context too.

Definition 2 (UFC coverage). *Given a linear temporal formula φ, a condition $a \in AC(\varphi)$ is the* unique first cause *(UFC) for φ along a path π, or equivalently, π is an adequate (UFC positive)* test case *for a in φ, if $\pi \models \varphi$ and there is an a-variant π' of π such that $\pi' \not\models \varphi$.*

When φ is a LTL formula, one can build a *trap formula* $ufc(a, \varphi)$, which is a *temporal* formula characterising adequate test cases for a in φ, i.e. paths on which a is UFC for φ.[6] $ufc(a, \varphi)$ is defined by induction on φ. For example, given $a \in AC(\varphi_a)$:

$$\begin{aligned} ufc(a, a) = a; ufc(a, \neg a) &= \neg a \\ ufc(a, \varphi_a \vee \varphi_b) &= ufc(a, \varphi_a) \wedge \neg\varphi_b \\ ufc(a, \mathsf{F}\,\varphi_a) &= (\neg\varphi_a)\,\mathsf{U}\,ufc(a, \varphi_a) \\ ufc(a, \mathsf{G}\,\varphi_a) &= \varphi_a\,\mathsf{U}\,(ufc(a, \varphi_a) \wedge \mathsf{G}\,\varphi_a) \end{aligned}$$

A complete definition is found in [6], and as a refined version later in this section. This characterisation of test cases for LTL only applies to complete, *infinite* paths. Realistic testing practices, however, are inherently limited to finite, *truncated* paths. In this context, the test case coverage criteria need to be refined further. Consider, for instance, the formula $\varphi = \mathsf{G}\,(a \vee b)$ expressing the requirement that either a or b must hold at any time. According to the definition above, an adequate test case for a in φ must satisfy

$$ufc(a, \varphi) = (a \vee b)\,\mathsf{U}\,((a \wedge \neg b) \wedge \mathsf{G}\,(a \vee b))$$

A concrete, finite test case π_f may reach a point where $a \wedge \neg b$, showing evidence that a *may* contribute to making φ true. However, there is no guarantee that this π_f is indeed a prefix of a π that satisfies φ, that is, that $a \vee b$ can hold indefinitely beyond the end of π_f. Such a finite prefix is defined as a *weak* test case for a in $\mathsf{G}\,(a \vee b)$.

A comprehensive treatment of temporal logic on truncated paths is found in [13], where strong and weak variants of semantic relations on finite prefixes are defined ($\pi_f \models^+ \varphi$ and $\pi_f \models^- \varphi$, respectively), where $\pi_f \models^- \varphi$ iff $\pi_f \not\models^+ \neg\varphi$. Intuitively, $\pi_f \models^+ \varphi$ iff π_f "carries all necessary evidence for" φ, whereas $\pi_f \models^- \varphi$ iff it "carries no evidence against" φ. In particular, if $\pi_f \models^+ \varphi$, then for any (non-truncated) π extending π_f we have $\pi \models \varphi$. Furthermore, [6] defines strengthening and weakening transformations $[\varphi]^+$ and $[\varphi]^-$ such that $\pi_f \models^\pm \varphi$ iff $\pi_f \models [\varphi]^\pm$.[7] In essence, $[\varphi]^+$ converts weak untils to strong untils, and vice-versa for $[\varphi]^-$; in particular, $[\mathsf{G}\,\varphi]^+ = \bot$ and $[\mathsf{F}\,\varphi]^- = \top$.

On this basis, [6] adapts the notion of UFC coverage by requiring that a (finite) test π_f case for a in φ satisfies φ according to the *standard* semantics up

[6] [6] uses a different notation φ^+ for the set of (positive) trap formulae for conditions in φ, that is, $\varphi^+ = \{ufc(a, \varphi) \mid a \text{ occurs in } \varphi\}$. The notation $\varphi^- = (\neg\varphi)^+$ is also defined. We do not use these notations here to avoid confusion with strong/weak semantic variants $\models^+$ and $\models^-$ (see below).

[7] Denoted $strong[\varphi]$ and $weak[\varphi]$ in [6], using LTL semantics extended to finite traces as in [13].

to the point where the effect of a is shown, and according to the *weak* semantics thereafter. For example, the trap formula for $\varphi = \mathsf{G}\,(a \Rightarrow \mathsf{F}\,b)$ becomes

$$\begin{aligned} \mathit{ufc}(a,\varphi) &= (a \Rightarrow \mathsf{F}\,b)\;\mathsf{U}\;((\neg a \wedge \neg\mathsf{F}\,b) \wedge [\mathsf{G}\,(a \Rightarrow \mathsf{F}\,b)]^{-}) \\ &= (a \Rightarrow \mathsf{F}\,b)\;\mathsf{U}\;(\neg a \wedge \neg\mathsf{F}\,b) \end{aligned}$$

since $[\mathsf{G}\,(a \Rightarrow \mathsf{F}\,b)]^{-}$ reduces to $\top$. This also illustrates a lack of uniformity in the approach: the weakening cancels *some* of the obligations on $\mathsf{F}\,b$, but not all, although the truncation may equally prevent all of them of being reached. Instead, in this paper we keep both weak and strong interpretations and apply them uniformly, obtaining two refined variants of UFC coverage.

Definition 3 (Strong/weak UFC coverage). *Given a linear temporal formula φ, a condition $a \in AC(\varphi)$ is the* strong *(resp.* weak*) unique first cause for φ along a finite path π_f, or equivalently, π_f is an adequate* strong *(resp.* weak*)* test case *for a in φ, if $\pi_f \models^{+} \varphi$ (resp. $\models^{-}$) and there is an a-variant π'_f of π_f such that $\pi'_f \not\models^{+} \varphi$ (resp. $\models^{-}$).*

As an example, the prefix in Figure 2 is, at the same time, a strong test case for a in $\mathsf{F}\,(a \wedge \neg b)$ and a weak test case for a in $\mathsf{G}\,(a \vee b)$. Observe that, consistently the discussion above, no finite prefix can strongly test a formula such as $\mathsf{G}\,a$, whose negation contains eventualities. We then refine $\mathit{ufc}(a,\varphi)$ into strong and

Fig. 2. Strong test case for a in $\mathsf{F}\,(a \wedge \neg b)$ and weak test case for a in $\mathsf{G}\,(a \vee b)$.

weak variants ufc^{+} and ufc^{-}, such that $\pi_f \models \mathit{ufc}^{\pm}(a,\varphi)$ iff π_f is a strong/weak test case for a in φ. These are jointly defined as follows, given $a \in AC(\varphi_a)$ and $b \in AC(\varphi_b)$:[8]

$$\begin{aligned} \mathit{ufc}^{\pm}(a,a) &= a \;\;;\;\; \mathit{ufc}^{\pm}(a,\neg a) = \neg a \\ \mathit{ufc}^{\pm}(a,\varphi_a \wedge \varphi_b) &= \mathit{ufc}^{\pm}(a,\varphi_a) \wedge [\varphi_b]^{\pm} \\ \mathit{ufc}^{\pm}(a,\varphi_a \vee \varphi_b) &= \mathit{ufc}^{\pm}(a,\varphi_a) \wedge [\neg\varphi_b]^{\pm} \\ \mathit{ufc}^{\pm}(a,\mathsf{X}\,\varphi_a) &= \mathsf{X}\;\mathit{ufc}^{\pm}(a,\varphi_a) \\ \mathit{ufc}^{\pm}(a,\mathsf{F}\,\varphi_a) &= [\neg\varphi_a]^{\pm}\;\mathsf{U}\;\mathit{ufc}^{\pm}(a,\varphi_a) \\ \mathit{ufc}^{\pm}(a,\mathsf{G}\,\varphi_a) &= [\varphi_a]^{\pm}\;\mathsf{U}\;(\mathit{ufc}^{\pm}(a,\varphi_a) \wedge [\mathsf{G}\,\varphi_a]^{\pm}) \\ \mathit{ufc}^{\pm}(a,\varphi_a\;\mathsf{U}\;\varphi_b) &= [\varphi_a \wedge \neg\varphi_b]^{\pm}\;\mathsf{U}\;(\mathit{ufc}^{\pm}(a,\varphi_a) \wedge [\neg\varphi_b]^{\pm} \wedge [\varphi_a\;\mathsf{U}\;\varphi_b]^{\pm}) \\ \mathit{ufc}^{\pm}(b,\varphi_a\;\mathsf{U}\;\varphi_b) &= [\varphi_a \wedge \neg\varphi_b]^{\pm}\;\mathsf{U}\;\mathit{ufc}^{\pm}(b,\varphi_b) \\ \mathit{ufc}^{\pm}(a,\varphi_a\;\mathsf{W}\;\varphi_b) &= [\varphi_a \wedge \neg\varphi_b]^{\pm}\;\mathsf{U}\;(\mathit{ufc}^{\pm}(a,\varphi_a) \wedge [\neg\varphi_b]^{\pm} \wedge [\varphi_a\;\mathsf{W}\;\varphi_b]^{\pm}) \\ \mathit{ufc}^{\pm}(b,\varphi_a\;\mathsf{W}\;\varphi_b) &= [\varphi_a \wedge \neg\varphi_b]^{\pm}\;\mathsf{U}\;\mathit{ufc}^{\pm}(b,\varphi_b) \end{aligned}$$

[8] This definition covers all cases, by pushing negations down to atoms and by symmetry of Boolean operators. Cases for F and G could be derived from U and W.

Specifically, sub-terms in G and W are strengthened to $\bot$ and U in the +-cases; in particular, $ufc^{+}(a, \mathsf{G}\varphi)$ boils down to $\bot$ for any φ (not a tautology), reflecting the fact that no adequate strong (finite) test case exists for $\mathsf{G}\varphi$. Given an atomic condition a appearing in a formula φ and an execution model M, if there is a *strong* test case $\pi_f \models ufc^{+}(a, \varphi)$ in the traces of M, then π_f shows that a can *necessarily* positively affect φ, in the sense that any extension π of π_f indeed satisfies φ. On the other hand, if $\pi_f \models ufc^{-}(a, \varphi)$, then π_f only shows that a can *possibly* positively affect φ, in the sense that there is no guarantee that this prefix can be completed to a full path of M that satisfies φ. It is also possible that there is no π_f in M for which $\pi_f \models ufc^{\pm}(a, \varphi)$: if φ is a positive (desired) property, then this means that a is a *vacuous* condition in φ w.r.t. M [7]; if φ is a negative (forbidden) property, then it confirms that this particular case of φ cannot happen, which is the desired result. A test fails if it is possible to find a path π_f in M such that $\pi_f \models ufc^{\pm}(a, \varphi)$, where φ is a negative property.

3 Planning

> *[...] planning can be considered a process of generating descriptions of how to operate some system to accomplish something. The resulting descriptions are called* **plans***, and the desired accomplishments are called* **goals***. In order to generate plans for a given system a* **model** *of how the system works must be given.*" [3]

Traditionally, in the description of a system there is a distinction between *states* and *actions* (see, for instance, the STRIPS planner and its associated language [14]). In this paper, however, we take a different approach and we consider the Europa 2 planner [3]; Europa 2 produces finite horizon, deterministic plans. The key concept of Europa 2 is the one of *tokens*, i.e., a temporally scoped (true) fact. For instance, a printer being ready between the time $t = 2$ and $t = 5$ is represented using a token `ready` (see the first token in Figure 3). Tokens may represent states of a single object in the system, and are sometimes

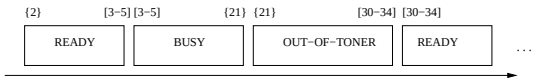

Fig. 3. A sequence of tokens in a timeline.

mutually exclusive. A *timeline* is a structure where sequences of tokens appear contiguously. For instance, the state of a printer can be described by a sequence of tokens, as in Figure 3 (therefore, state is a timeline). In this example, the syntax $\{2\}$ denotes the time instant $t = 2$, while $[3 - 5]$ denotes a value of t between 3 and 5.

Europa 2 allows for the expression of various relations between tokens, based on Allen's interval logic [15]. Allen's interval logic includes 13 possible relations between a pair of tokens: `meets/met_by, before/after, equals, starts/ends,`

`contains/contained_by, contains_start/starts_during, contains_end/ends_during`.
A planning problem is formulated in Europa 2 as a set of tokens (possibly belonging in a timeline), a set of rules expressed using the relations presented above, and a set of *goals*: these elements define a so-called partial plan, which is refined by Europa 2 into a complete plan, i.e., a plan where all tokens are active and no unbound variables exist. We refer to [3] and references therein for further details about the theoretical foundations of Europa 2. The input files of Europa 2 are expressed using the New Domain Description Language (NDDL, pronounced "noodle"). NDDL is a (restricted) object-oriented, Java-like language; a simple example of a NDDL file (without goals) is given in Figure 4.

```
class SimplePrinter extends Timeline {
 predicate ready{} /* Predicates with no arguments and no body:
                      these are tokens */
 predicate busy{}
}
/* Simple rules to force a repeated cycle */
SimplePrintert::ready{
 eq(duration, 10);
 meets (object.busy);
 met_by(object.busy);
}
```

Fig. 4. A very simple NDDL file.

4 A concrete example: the Rover scenario

This section presents a concrete case study which will be used in the remainder of the paper to exemplify our methodology: an autonomous rover. A rover contains three main components: a navigator to manage the location and the movement of the rover; an instrument to perform scientific analysis; a commander, receiving instructions from scientists and directing the rover's operations. Each of these components is mapped into a class (respectively: `Navigator`, `Instrument`, and `Commands`); each of these classes is a timeline. Additionally, the domain contains two classes `Location` and `Path` to represent physical locations and paths between locations. Each of the timelines contain the following tokens:

- `Navigator`: `At, Going` (the rover is either at a location, or going between locations).
- `Instrument`: `stow, unstow, stowed`, `place`, `takesample` (the instrument can be stowed, can be in in the state of being stowed or unstowed, can be placed on a target, or it can take a sample).
- `Commands`: `takesample`, `phonehome`, `phonelander` (the rover can be instructed to take a sample, and it can communicate either with a lander, or directly with the Earth).

An extract of the NDDL code for this example is presented in Figure 5, where the NDDL code corresponding to a part of the declaration of the `Navigator` timeline is shown (notice the similarities with the syntax of Java). See the comments appearing in Figure 5 for more details. The class `Timeline` (not shown

```
class Location {
  string name;  int x;  int y;
   Location(string _name, int _x, int _y){
    name = _name;    x = _x;    y = _y;
  }
}
[...]
// Navigator is a Timeline
class Navigator extends Timeline {
 predicate At{
   Location location;
 }
 predicate Going{
   Location from;   Location to;
   neq(from, to);
     // This is a rule: it prevents rover from going from a location
     // straight back to that location.
 }
}
// This is a rule for the token At
Navigator::At{
  met_by(object.Going from);// Allen's relation: each At is met_by
                            // a Going token
  eq(from.to, location); // next Going token starts at this location
  meets(object.Going to);  // Allen's relation: each At meets
                           // a Going token
  eq(to.from, location); // prevous Going token ends at this location
}
[...]
// A possible goal
goal(Commands.TakeSample sample); sample.start.specify(50);
sample.rock.specify(rock4);
```

Fig. 5. Excerpt from the NDDL file for the rover example.

in the example) is a super-class containing two variables `start` and `end` (notice that in NDDL all variables and methods are public). A possible goal for this scenario is presented at the end of the code in Figure 5: the goal is to begin a sample of `rock4` at time 50.

When a NDDL file encoding the scenario and the goals is passed to Europa 2, the planner generates a plan, similarly to the one presented in Figure 6

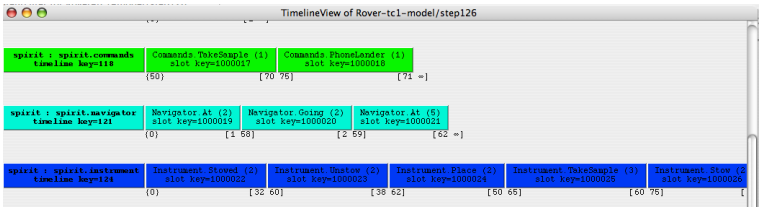

Fig. 6. Generated plan.

5 Flight rules

Flight Rules are requirements that must be satisfied in every execution. Typically, flight rules are a plain text document which accompanies the description of scenarios. For instance, a flight rule for the example presented in Section 4 is "*all Instruments must be stowed when moving*". The majority of flight rules falls into one of the *temporal patterns* defined in [16] and thus can be encoded by means of an LTL formula. For instance, the flight rule above can be encoded as $\varphi = G(p \rightarrow q) = G(\neg p \vee q)$, where $p =$ moving and $q =$ stowed.

5.1 Coverage sets for flight rules

As flight rules can be encoded as LTL formulae, the methodology presented in Section 2 can be applied to generate a set of test cases for flight rules *with coverage guarantees*. As an example, we consider the flight rule presented above, namely $\varphi = G(\neg p \vee q)$ (where p = moving and q = stowed). Being a safety formula, we can only have *weak* evidences for the positive test cases (see Section 2.2) because the planner can only generate finite executions. More in detail, we have the following three test cases:

1. $ufc^{-}(q, \varphi) = ((\neg p \wedge q)U(\neg p \wedge \neg q))$;
2. $ufc^{-}(p, \varphi) = ((\neg p \wedge q)U(p \wedge q \wedge \varphi))$;
3. $ufc^{+/-}(p, \varphi) = ufc^{+/-}(q, \varphi) = ((\neg p \vee q)U(p \wedge \neg q))$.

The first positive test case tests the true value of the whole formula caused by the proposition q (i.e., stowed); the second test case tests the proposition p. There is only one test case for false value of the formula and it is contributed by both propositions; notice that this test case is the same for weak and for strong evidence.

A similar exercise could be repeated for all the flight rules appearing in the specification for any given scenarios. Using the methodology presented above to compute test cases *with UFC coverage guarantees* would be an improvement *per se* with respect to the current testing methodologies (currently, test cases for Europa 2 are generated manually and without coverage guarantees). But our approach can be refined further, to the benefit of plan developers: in the next section we present how the execution of tests can be automated by translating temporal formulae into planning goals.

6 From temporal formulae to planning goals

The key idea of this section is to translate LTL formulae encoding test cases into planning goals. This is achieved by building a parse tree of the formula and by associating a timeline to each node of the parse tree.

We present the details of this methodology using the first positive test case for the scenario presented in Section 4, namely, for the formula $(\neg p \wedge q)U(\neg p \wedge \neg q)$, where $p =$ moving and $q =$ stowed

We start with proposition p, which is true whenever the rover is moving. We define a new timeline `prop-p` containing the two tokens `TRUE` and `FALSE`: token `TRUE` of `prop-p` is the case when the token `Going` of `Navigator` holds, and token `FALSE` of the timeline `prop-p` holds when `TRUE` does not hold. These requirements are translated into the NDDL code represented in Figure 7 (top part). The negation of proposition p is defined as a new timeline `prop-not-p` composed by the two tokens `TRUE` and `FALSE` and by the rules presented in Figure 7 (bottom part).

Proposition q (representing the fact that instruments are stowed) and its negation are defined in a similar way as new timelines. The conjunction of two propositions is encoded as a new timeline with four tokens representing the

```
// The new timeline with two tokens:
class prop-p extends Timeline {
    predicate TRUE { };
    predicate FLASE { };
}
// The rule for TRUE:
prop-p::TRUE {
    Navigator nav;
    equals(nav.Going);
    met_by(object.FALSE f1);
    meets(object.FALSE f2);
}
// Additional rule for Navigator::Going
Navigator::Going {
  prop-p p;
  equals(p.TRUE);
}

class prop-not-p {
    predicate TRUE { };
    predicate FLASE { };
}
prop-not-p::TRUE {
  meets(object.FALSE f1);
  met_by(object.FALSE f2);
  prop-p p;
  equals(p.FALSE);
}
```

Fig. 7. NDDL code for proposition p and $\neg p$ (moving).

possible truth values of the two conjuncts. The scope of each token is defined using the two Allen's relations `contains` and `contained_by`. The truth value of the whole conjunction is obtained using a third timeline with two tokens only (`TRUE` and `FALSE`). The NDDL code corresponding to the conjunction of two propositions is available from the authors upon request. We are now in a position to test the formula $\varphi = (\neg p \wedge q) U (\neg p \wedge \neg q)$. For simplicity, let $\varphi = A \mathsf{U} B$, where A and B are encoded using the timelines `prop-A` and `prop-B` respectively. The LTL proposition φ holds in the model iff the following goal can be satisfied:

```
goal(prop-A.TRUE); goal(prop-B.TRUE); eq(prop-A.start,0);
contains_end(prop-B.TRUE,prop-A.TRUE);
```

Intuitively, the goal above states that proposition A has to be true at the beginning of the run (`eq(prop-A.start,0)`) and that B becomes true *before* the end of `TRUE` of A (`contains_end(prop-B.TRUE,prop-A.TRUE)`). The additional NDDL code presented above is added to the original NDDL code for the scenario (notice that, in doing so, the instrumentation process cannot introduce bugs in the original model). The new "enriched" NDDL code is passed to Europa 2 for plan generation. If a plan can be obtained with the additional constraints for the positive test case, the test is passed successfully. Figure 8 depicts Europa 2 output for the first test case. Notice the additional timelines for the propositions Boolean propositions (compare with Figure 6). This plan illustrates an execution where the atomic condition q (stowed) is the unique first cause. This exercise can be repeated for the second positive test case, which is passed, and for the negative test case. As expected, no plan can be generated for the negative test case.

6.1 Discussion

While the scenario presented above is not as rich as real production and mission environments, it is nevertheless more complex than the biggest examples that could be

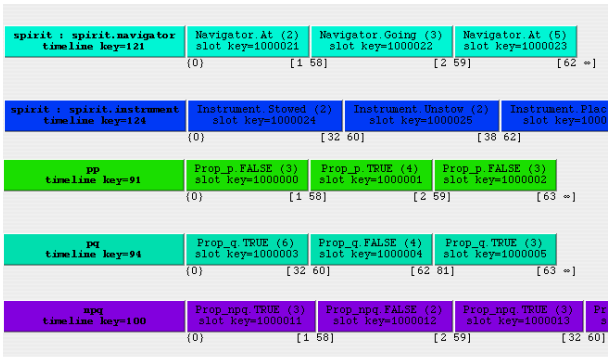

Fig. 8. Generated plan with additional timelines.

analysed using translations into model checkers [5, 4]. We have run our tests on a standard machine and the introduction of the new timelines did not affect the performance of the planner for positive test cases. This result was expected, as a domain with additional constraints should be "easier" to solve than a less constrained domain: the introduction of the new timelines seems to balance this benefit. Negative test cases, however, require more computational power because of the backtracking involved in over-constrained domains. The planner eventually fails on negative test cases in around 10 minutes for the rover example, while it is able to produce a result in less than 30 seconds for positive test cases. Even though our aim in this paper is mainly to provide *feasible* coverage guarantees for test suites of planning domains and we are not concerned with performance issues, nevertheless we consider our preliminary results encouraging.

7 Conclusion

Traditionally, the problem of verifying planning domains has been approached by translating the planning domain into an appropriate formalism for a model checker, where verification can be performed either in a direct way, or by generating test cases, with the exception of [17] where a planning techniques are suggested for the generation of tests. This latter work differs from ours in that different coverage conditions are considered, and tests are not generated from flight rules (i.e., temporal specifications). Some issues remain to be investigated. For instance, we do not have a methodology to deal with the translation of nested temporal operators into planning goals (but we did not find nested temporal operators in the flight rules analysed). We are currently working on this issue and on a software tool to automate the methodology: we are implementing a parser from temporal patterns encoding flight rules to LTL trap formulae using the definitions in Section 2.2, and finally to NDDL code.

References

1. S. W. Squyres et al.: The Opportunity Rover's Athena Science Investigation at Meridiani Planum, Mars. Science (2004) 1698–1703
2. S. W. Squyres et al.: The Spirit Rover's Athena Science Investigation at Gusev Crater, Mars. Science (2004) 1698–1703

3. McGann, C.: How to solve it (Europa 2 User Guide). NASA Ames report (2006)
4. Khatib, L., Muscettola, N., Havelund, K.: Verification of plan models using UPPAAL. Lecture Notes in Computer Science **1871** (2001)
5. Penix, J., Pecheur, C., Havelund, K.: Using Model Checking to Validate AI Planner Domains. In: Proceedings of the 23rd Annual Software Engineering Workshop, NASA Goddard (1998)
6. Whalen, M.W., Rajan, A., Heimdahl, M., Miller, S.P.: Coverage metrics for requirements-based testing. In: Proceedings of ISSTA'06, New York, NY, USA, ACM Press (2006) 25–36
7. Tan, L., Sokolsky, O., Lee, I.: Specification-based testing with linear temporal logic. In: Proceedings of IRI04, IEEE Society (2004)
8. Hong, H.S., Lee, I., Sokolsky, O., Ural, H.: A temporal logic based theory of test coverage and generation. In: Proceedings of TACAS '02, London, UK, Springer-Verlag (2002) 327–341
9. Beizer, B.: Software testing techniques (2nd ed.). Van Nostrand Reinhold Co., New York, NY, USA (1990)
10. Hayhurst, K.J., Veerhusen, D.S., Chilenski, J., Riersn, L.K.: A practical tutorial on modified condition/decision coverage. Technical Report TM-2001-210876, NASA Langley Research Center (2001)
11. RTCA: Software Considerations in Airborne Systems and Equipment Certification. (1992)
12. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge, Massachusetts (1999)
13. Eisner, C., et al.: Reasoning with temporal logic on truncated paths. In: Proceedings of CAV '03. Volume 2725 of LNCS., Springer (2003)
14. Fikes, R., Nilsson, N.J.: Strips: A new approach to the application of theorem proving to problem solving. Artificial Intelligence **2**(3/4) (1971) 189–208
15. Allen, J.F.: Maintaining knowledge about temporal intervals. Commun. ACM **26**(11) (1983) 832–843
16. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Property specification patterns for finite-state verification. In Ardis, M., ed.: Proceedings of FMSP'98, New York, ACM Press (1998) 7–15
17. Howe, A.E., Mayrhauser, A. von, Mraz, R.T.: Test Case Generation as an AI Planning Problem. In Journal of Automated Software Engineering, 4:1, 1997, 77-106