# PDVer, a Tool to Verify PDDL Planning Domains

**Franco Raimondi**
Department of Computer Science, UCL
London, UK
f.raimondi@cs.ucl.ac.uk

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

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

**Abstract**

We present a methodology and a tool for the problem of testing and verifying that a PDDL planning domain satisfies a set of requirements, a need that arises for instance in space missions. We first review and analyse coverage conditions for requirement-based testing, and present how test cases can be derived automatically from requirements. Additionally, we show how test cases can be translated into additional planning goals. To automate this process, we introduce PDVer, an Eclipse plug-in for the automatic generation of PDDL code from requirements expressed in LTL. We evaluate the effectiveness of our approach and the usability of our tool against the Rovers domain from the fifth International Planning Competition (IPC-5).

## 1 Introduction

A number of complex systems currently deployed present a significant amount of autonomy, as in the case of the NASA rovers Spirit and Opportunity (S. W. Squyres et al. 2004a; 2004b) exploring the surface of Mars since January 2004. Typically, these system include some kind of planning infrastructure to take appropriate actions *autonomously*. However, 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 *safety conditions* can be imposed to avoid damages and to minimize the risk of failures, such as "all scientific instruments must be appropriately stored when the rover is moving" or "if the rover is at a given rock, then it must send a picture of the rock". These kind of conditions are called *flight rules* and affect various stages of system development, from design to deployment, including the verification that planning domains do not violate such requirements.

The aim of this paper is to investigate the problem of verifying planning domains written in the Planning Domain Definition Language (PDDL (Gerevini and Long 1997)). More in detail, we investigate the use of testing methods for the verification of flight rules. We first consider MC/DC coverage (Hayhurst *et al.* 2001), a metric well known to mission engineers, then we review our own extension (Pecheur *et al.* 2009) of MC/DC to temporal patterns. Our extension provides a *formal* background to the notion of coverage for *requirements-based testing*, extending the work of (Whalen *et al.* 2006; Tan *et al.* 2004; Hong *et al.* 2002). Our choice is motivated by the fact that actual developers are familiar with MC/DC and are interested in the possibility of stressing particular conditions in a given requirement. We consider testing methodologies instead of trying to encode directly a PDDL domain into the language of a model checker for a number of reasons:

- the size of the state space may be too large to be analysed exhaustively with a model checker, but it may be still explorable partially by a planner;
- the PDDL model of the system could include features (such as durations and costs) that are hard to encode in the input language of a model checker (see, for instance, the problem of translating planning models into an adequate input for a model checker presented in (Penix *et al.* 1998; Khatib *et al.* 2001));
- 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, reporting the formula true. In some cases, planning engineers are interested in "stressing" a particular atomic proposition in a formula, and make a formula true *because of* that particular proposition.

This paper makes two contributions:

1. we illustrate how to encode test cases using PDDL, thereby enabling the *use of the planner itself to perform verification*: essentially, the verification step is reduced to a planning problem with additional constraints.
2. We introduce PDVer, a tool that allows the *automatic* generation of test cases from LTL specifications and produces PDDL output, thus providing a concrete solution for the verification problem of planning domains. PDVer makes use of the coverage of requirements mentioned above to provide a complete set of test cases.

Verification of planning domains has been investigated in the past, for instance in (Khatib *et al.* 2001; Penix *et al.* 1998). 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: 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.

The rest of the paper is organised as follows: we review MC/DC coverage and PDDL in Section and we introduce coverage metrics for temporal specification in Section 2 ; we present the PDVer tool in Section 3 , and we provide a concrete example in Section 4.

## 2 Background

### MC/DC coverage and requirement-based testing

Various metrics exist to quantify the coverage of test suites (Beizer 1990), particularly for *structural* testing. In this section we briefly review MC/DC (structural) coverage. MC/DC coverage is required for the most critical categories of avionic software (RTCA 1992) 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:

| $a$ | $b$ | $a \vee b$ |
|---|---|---|
| $\top$ | $\bot$ | $\top$ |
| $\bot$ | $\top$ | $\top$ |
| $\bot$ | $\bot$ | $\bot$ |

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

There is some flexibility in how "independently affect" is to be interpreted, see (Chilenski and Miller 1994; Hayhurst *et al.* 2001; Chilenski 2001). The original definition in (RTCA 1992) 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 (Hayhurst *et al.* 2001) defines a weaker version based on logic gate networks, called *masking* MC/DC.

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 $a$ are $a \wedge \neg b$ and $\neg a \wedge \neg b$.

### The Planning Domain Definition Language

PDDL (Gerevini and Long 1997) is a language for the definition of planning domains and problems, developed by the model-based planning community as a standard language for planning competitions. PDDL supports the definition of domain models and *problems*, where a problem is a set of goals and constraints that define one planning problem with respect to a given model. The latest version, PDDL 3.0, supports a limited form of temporal logic constraints as part of the problem. The following temporal primitives are supported:

```
<GD> ::= (at-end <GD>) |
         (always <GD>) |
         (sometime <GD>) |
         (within <num> <GD>) |
         (at-most-once <GD>) |
         (sometime-after <GD>) |
         (sometime-before <GD>) |
         (always-within <num> <GD> <GD>) |
         (hold-during <num> <num> <GD>) |
         (hold-after <num> <GD>)
```

where `<num>` is a numeric literal denoting time constraints. Constraints are interpreted over finite sequences of states labelled with time (PDDL semantics is formalised in (Gerevini and Long 1997)). The timing aspects are not used in this paper. We are interested in the following PDDL constraints operators, with the corresponding translation into Linear Temporal Logic (LTL, we refer to (Clarke *et al.* 1999) for more details):

$$\begin{aligned} \text{"}always\text{"}(\varphi) &= \mathsf{G}\,\varphi \\ \text{"}sometime\text{"}(\varphi) &= \mathsf{F}\,\varphi \\ \text{"}sometime-before\text{"}(\varphi,\psi) &= (\neg\varphi \wedge \neg\psi)\,\mathsf{W}\,(\psi \wedge \neg\varphi) \\ &= \neg\varphi\,\mathsf{W}\,(\psi \wedge \neg\varphi) \end{aligned}$$

where $\varphi_1\,\mathsf{W}\,\varphi_2$ ("$\varphi_1$ unless $\varphi_2$") is true iff $\varphi_1$ holds at least as long as $\varphi_2$ does not hold . Let $\varphi = \neg\varphi_1 \wedge \neg\varphi_2$ and $\psi = \varphi_2$, then $\text{"}sometime-before\text{"}(\varphi,\psi) = (\varphi_1 \vee \varphi_2)\,\mathsf{W}\,\varphi_2 = \varphi_1\,\mathsf{W}\,\varphi_2$ and thus

$$\varphi_1\,\mathsf{W}\,\varphi_2 \quad = \quad \text{"}sometime-before\text{"}(\neg\varphi_1 \wedge \neg\varphi_2, \varphi_2)$$

Given the above, we have all the expressivity needed to translate LTL to PDDL constraints. The current PDDL definition does not allow nested temporal modalities in constraints: that is planned as a future extension, although using this feature is likely to limit the set of planners that will be able to support the corresponding verification.

## Coverage of requirements

In this section we define what is an "adequate" test case for a condition (i.e., an atomic proposition) $a$ in a LTL formula $\varphi$. We give here only the key concepts and we refer to (Pecheur *et al.* 2009) for further details.

Consider a LTL formula $\varphi$, interpreted over (finite or infinite) paths $\pi$, built from a set of states $S$. 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 $\pi$.

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

$$\pi(AC(\varphi) - \{a\}) = \pi'(AC(\varphi) - \{a\})$$

Intuitively, an $a$-variant of a path $\pi$ is another path $\pi'$ such that the evaluation of all the conditions does not change, with the exception of the only condition $a$. The following definition provides a formal characterisation of *adequate test cases*:

**Definition 2** *An execution path $\pi$ is an adequate test case for an atom $a$ occurring in a formula $\varphi$ iff $\pi \models \varphi$ and there exists an $a$-variant $\pi'$ of $\pi$ such that $\pi' \not\models \varphi$. We denote with $\mathsf{FLIP}(\varphi, a)$ the set of all such paths.*

The intuition here is that a good test case for a condition $a$ in a formula $\varphi$ is a execution path $\pi$ such that $\varphi$ is true along that path and there exists (at least) another path $\pi'$ where "everything is the same" with the exception of $a$, and $\varphi$ is false on $\pi'$: this means that $a$ can *flip* the value of $\varphi$ (on $\pi$), i.e., $\varphi$ is true *because of* $a$.

Similarly to MC/DC, it is possible to characterise test cases by means of a trap formula. Given a LTL formula $\varphi$ and a condition $a \in AC(\varphi)$, we denote by $[\varphi]_a$ the trap formula encoding adequate test cases in the sense of Definition 2.

**Definition 3 Syntactic characterisation of trap formulae**

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

(where R is the standard "release" operator). Other cases are obtained by syntactic derivation:

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

These derivations for fixed point modalities are all of the form $[\varphi]_a = \varphi \wedge \varphi''$, where the recursive step occurs only in $\varphi''$. If required, and to avoid nesting of temporal operators for the PDDL translation, they can be rewritten into equivalent forms $[\varphi]_a = \varphi_1\,\mathsf{U}\,([\varphi_a]_a \wedge \varphi_2)$:

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

It is shown in (Pecheur *et al.* 2009) that $\pi \models [\varphi]_a$ if and only if $\pi \in \mathsf{FLIP}(\varphi, a)$. As an example, consider the formula $\mathsf{F}\,(a \vee b)$. 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

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

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 $a$.

## 3 From trap formulae to planning goals

As mentioned in Section 1, it is often necessary to verify planning domains against a set of requirements. Based on our observations at NASA Ames, the process of verifying that planning domains satisfy a set of flight rules is currently performed by engineers "manually", without coverage guarantees but based solely on engineers' expertise about the domain.

In this section we introduce PDVer, a tool that makes it possible to edit PDDL domains in a graphical editor and to derive test cases (in the form of new planning goals) from flight rules. PDVer is available from `http://www.cs.ucl.ac.uk/staff/f.raimondi/pddleditor_1.0.0.jar` and it has been developed as a plug-in for Eclipse 3.3. A single .jar file is available to download and it is installed by simply dropping the file in the `plugin/` directory of Eclipse. PDVer uses the ANTLR parser to generate lexers and parsers for LTL and PDDL, and a tree grammar for PDDL. The source code can be found in the src/ directory. It is divided into various packages:

- An LTL parser, syntax analyser, and AST (Abstract Syntax Tree) manipulator for LTL. In particular, this component implements the algorithms to compute the trap formulae as described in Section 2.

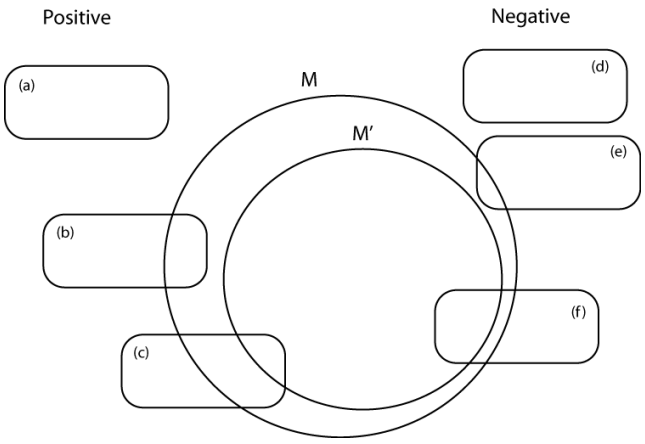


Figure 3: Summary of coverage cases for positive and negative test cases

- A PDDL parser. This parser is used to obtain the AST representation of the domain, in order to perform checks on the LTL formulae provided, to translate the trap formulae from LTL to PDDL, and to appropriately quantify free variables (see below); furthermore, it is used to perform syntax highlighting.
- A plug-in specific component that contains the Eclipse-specific Java code for plug-in generation. The plug-in is activated when clicking on a .pddl file in the navigator or package explorer. The plug-in is implemented as a graphical editor.

The editor window is depicted in Figure 1. By selecting the "Generate tests" tab (see red circle in the figure above) it is possible to access the section for test generation (see Figure 2). As an example, consider the formula: `"(at_place rover0 ?x)" -> F("(have_rock_analysis rover0 ?x)")`. The new PDDL goals encoding the various test cases are generated by typing this formula in the appropriate text box and by selecting the "Generate PDDL" button. The generated PDDL code is displayed in the text window (Figure 2), from where it can be selected and pasted back in the domain definition in the editor tab and then passed to the planner. In this way, *the planner itself can be used to perform verification*.

### Discussion

PDver enables the generation of new planning goals from flight rules, which can then be used in conjunction with the original domain and passed a planner. However, care must be taken to interpret the results of this process that can be seen as "testing as planning".

For a "positive" test (i.e., when it is expected that the tested property holds), the planner should be able to produce a plan with the additional constraints. Alternatively, for a negative test the planner should not be able to produce a test. However, these results are not necessary and a number of possibilities exists, as reported in Figure 3.

A first issue to consider is the completeness of the planner. We assume that planners are sound (i.e. they will not produce a plan which is invalid wrt the PDDL input), but planners may not be complete, i.e., they may fail to produce a plan when in fact it exists. The failure to produce a plan can result in non-termination errors such as in timeouts or out-of-memory errors or, in the worst case, with termination without a plan. Therefore, the behaviour of the planner should be investigated and understood to validate testing results. In Figure 3 we denote the non-completeness of the planner by means of two circles: the external circle M denotes the set of plans that are consistent with a given PDDL domain description. The internal circle M' denotes the set of plans that can be potentially generated by the planner. Let the boxes denote the set of plans that are compatible with a given set of constraints. Six situations can occur:

**(a)** corresponds to a positive test case that has no intersection with the domain. For instance, a test case in this category could impose constraints on variables that are not allowed by the domain. In the case of FLIP, this means that it is not possible to exercise the effect of an atom on a given formula.

**(b)** corresponds to a positive test cases that, in theory, should result in a valid plan. However, as a consequence of the fact that the planner is not complete, the planner fails to find a valid plan.

**(c)** corresponds to a positive test case that can successfully be covered by the planner.

**(d)** corresponds to a negative test case. If the planner terminates without a valid plan it confirms, as expected, that the negative test case cannot be covered.

**(e)** corresponds to a negative test case that should be covered by the planner (i.e., the test should *fail* by showing that an un-expected plan exists). This is potentially the most critical situation: if the planner does not terminate, then it is not possible to guarantee that the domain does not allow unwanted behaviours. On the other hand, if the planner does terminate, then the situation could be misunderstood for case **(d)** above.

**(f)** corresponds to a negative test case that can be covered by the planner. In this case, the existence of a plan reveals an error in the definition of the domain.

## 4 A concrete example

In this section we introduce an example of verification for some simple properties of a domain from the fifth International Planning Competition (IPC-5): a rover performing autonomous scientific exploration. The domain is characterised by various classes of objects, including rovers, waypoints, and a number of scientific instruments. Predicates are included to describe the various conditions, such as

`(at_place ?x - rover ?y - waypoint)`

to denote that a rover is at a given location, and

`(have_rock_analysis ?x - rover ?y - waypoint)`

to denote that a rover has analysed the rock at a waypoint. The full code for this domain is available from `http://zeus.ing.unibs.it/ipc-5/`. We employ

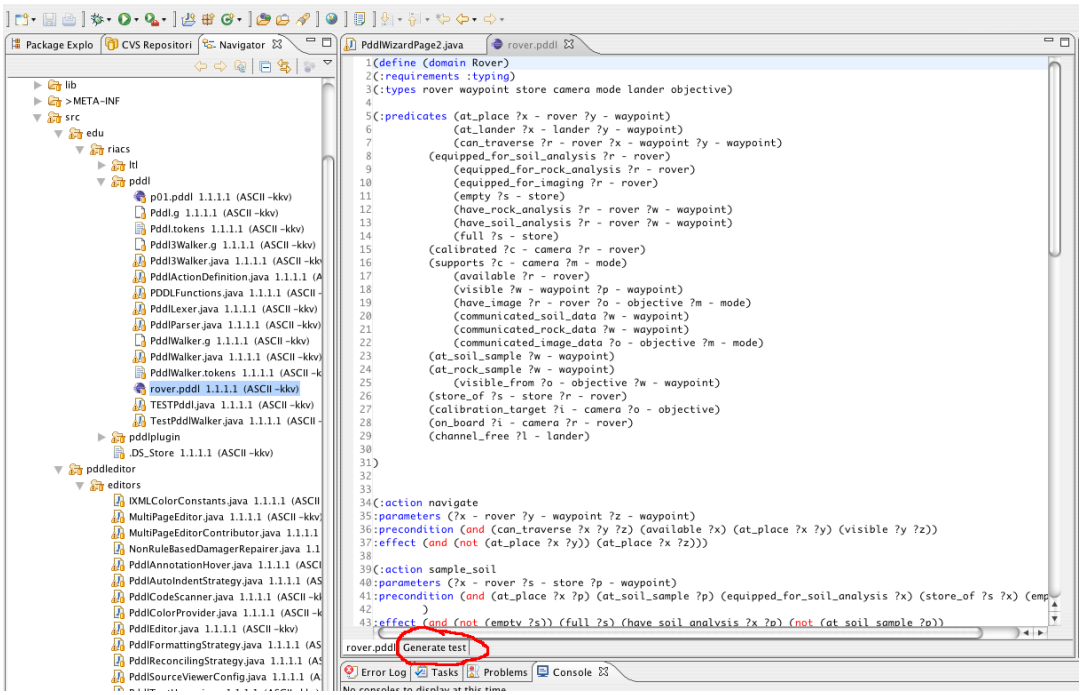


Figure 1: PDVer: editor window

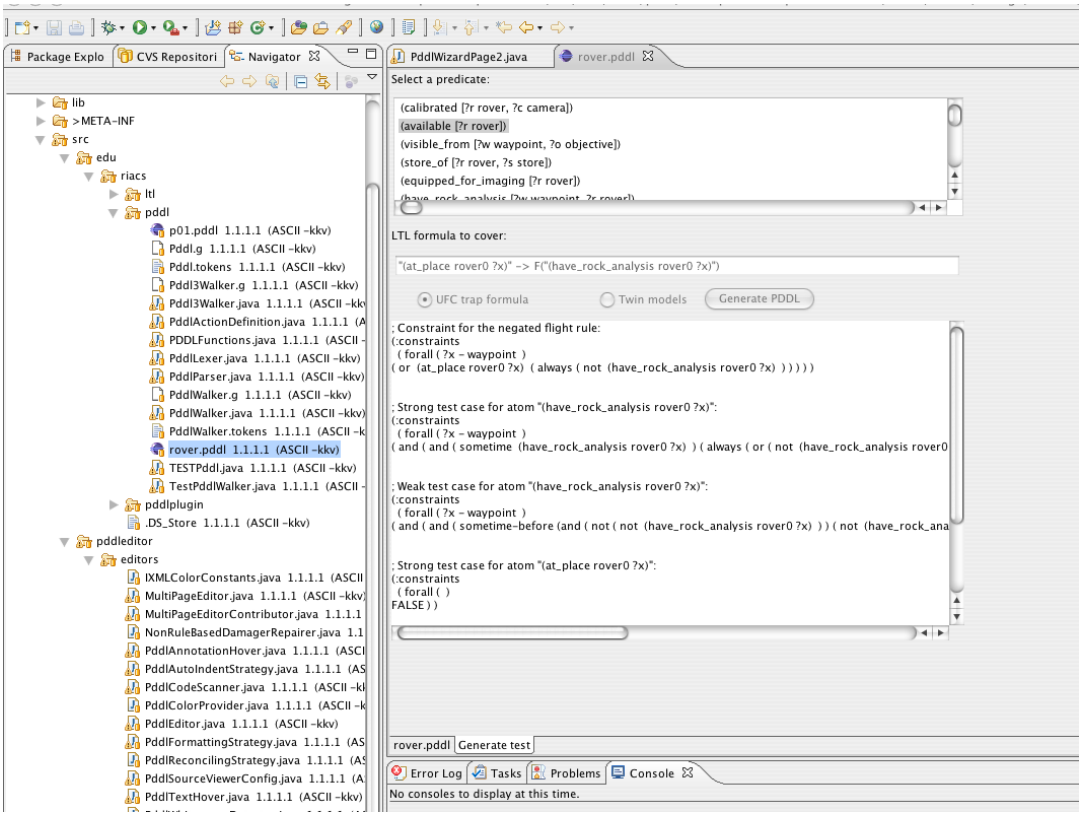


Figure 2: PDVer: test generation window

the MIPS-XXL planner (Edelkamp and Helmert 2001) to run our experiments for its support for PDDL 3.0.

- As a first example, consider the LTL formula

```
"(at_place rover0 ?x)" ->
  F("(have_rock_analysis rover0 ?x)")
```

  meaning that, if rover0 is at a certain place, then eventually in the future rover0 will have rock analysis for that place. PDVer generates various test cases from the LTL specification, including the following temporal goal[1]:

```
; Test case for atom "(at_place rover0 ?x)":
(:constraints
  ( forall ( ?x - waypoint  )
    ( and ( not  (at_place rover0 ?x)  )
          ( not ( sometime
        (have_rock_analysis rover0 ?x))))
))
```

  The original formula has the form $a \Rightarrow \mathsf{F}\ b$; the corresponding trap formula for atom $a$ is (see Section 2) $\neg a \wedge \neg \mathsf{F}\ (b)$, which is translated into the PDDL code reported above. Intuitively, this goal tries to exercise the atom `"(at_place rover0 ?x)"` in the LTL formula presented above, and the only way to do so is by requiring $b$ to be always false. Adding this constraint to the original domain results in MIPS-XXL failing to find a valid plan thus indicating that the test does not succeed (notice: as mentioned above this fact does not mean that there is an error in the original domain, but it simply means that it is not possible to exercise the atom `(at_place rover0 ?x)`, for all waypoints `x`, because the original problem does not allow the rover to be placed at all waypoints).

- However, if the free variable is removed and an actually visited waypoint is used:

```
"(at_place rover0 waypoint1)" ->
F("(have_rock_analysis rover0 waypoint1)")
```

  then the planner succeeds in producing a plan with the additional generated constraint because for this particular waypoint the domain allows the rover to be there.

- As a third example, consider the formula

```
"(have_soil_analysis rover0 ?w)" ->
     F("(communicated_soil_data ?w)")
```

  Meaning that if rover0 has the analysis of soil at a waypoint, then at some point that data will be communicated. The automatically generated test case for `(communicated_soil_data ?w)` fails in this case because there is no soil analysis for waypoint1 defined in the domain.

It is worth noting that the additional goals presented above do not change in a noticeable way the time required to generate a plan as constraints are *added* to the original domain. Moreover, no knowledge of coverage metrics is required to generate these tests, and the results can be easily interpreted with the use of Figure 3.

[1]PDVer accepts "free" variables, such as `x` in the following formula. When such a variable appears, the generated test case (i.e., the new goal) includes an universal quantifier).

## 5 Discussion and conclusion

Verification of planning domains has been investigated in the past mostly by reducing the verification problem to model checking. This process involves the translation of planning domains into a suitable input for model checkers (see (Penix *et al.* 1998; Khatib *et al.* 2001) and references therein), but it is often limited to simple examples with a limited state space.

In (Howe *et al.* 1997) the use of planning techniques is suggested for the generation of tests cases. This work differs from ours in that, first of all, the aim is not the verification of planning domains, but only the generation of test cases for other kind of domains. Additionally, different coverage conditions are considered, and tests are not generated from temporal specifications.

In this paper we have presented a different approach to the verification of planning domains: first we have illustrated how testing can be regarded as a *planning problem*, by translating LTL properties into new goals. Then, we have presented PDVer, a tool that automatically produces *planning goals* from requirements, with coverage guarantees. As an example, we have shown how to verify the property of a domain that can be solved by a state-of-the-art planner. To the best of our knowledge, PDVer is the first tool to support this verification methodology, with the additional benefit that the original domain does not need to be modified nor instrumented (new goals being added at the end of the original files), and scientists do not need to be familiar with the details of coverage metrics for requirements. Instead, test generation is automatic and results can be interpreted by means of Figure 3.

We are currently working on extensions of PDVer: PDDL 3 does not support nested temporal operators, and therefore only flight rules without nested temporal modalities can be translated into valid planning goals. To overcome this issue, we are building equivalent, non-nested expression for a number of requirements. Our final aim is to deliver PDVer to scientists developing plans for various missions, such as autonomous rovers, in-flight controllers, etc.

## References


B. Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

John Joseph Chilenski and Steven P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, pages 193–200, 1994.

John Joseph 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.

E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

Stefan Edelkamp and Malte Helmert. Mips: The model-checking integrated planning system. *AI Magazine*, 22(3):67–72, 2001.

A. Gerevini and D. Long. Plan constraints and preferences in PDDL3: The language of the fifth international planning competition, August 1997. Technical Report.

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

Hyoung Seok Hong, Insup Lee, Oleg Sokolsky, and Hasan Ural. A temporal logic based theory of test coverage and generation. In *TACAS 02*, pages 327–341, London, UK, 2002.

A.E. Howe, A. von Mayrhauser, and R.T. Mraz. Test case generation as an ai planning problem. *Journal of Automated Software Engineering*, 4:77–106, 1997.

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

C. Pecheur, F. Raimondi, and G. Brat. A formal analysis of requirements-based testing. In *Proceedings of ISSTA 2009*. ACM press, 2009.

J. Penix, C. Pecheur, and K. Havelund. Using Model Checking to Validate AI Planner Domains. In *Proceedings of the 23rd Annual Software Engineering Workshop*. NASA Goddard, 1998.

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

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

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

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.

M. W. Whalen, A. Rajan, M. P. E. Heimdahl, and S. P. Miller. Coverage metrics for requirements-based testing. In *ISSTA06*, pages 25–36, New York, NY, USA, 2006. ACM Press.