# Automatic Synthesis of Temporal Invariants

**Sara Bernardini**
London Knowledge Lab
23-29 Emerald Street, London WC1N 3QS
s.bernardini@lkl.ac.uk

**David E. Smith**
NASA Ames Research Center
Moffet Field, CA 94035–1000
david.smith@nasa.gov

### Abstract

We present a technique for automatically extracting temporal mutual exclusion invariants from PDDL2.2 planning instances. Our technique builds on other approaches to invariant synthesis presented in the literature, but departs from their limited focus on instantaneous discrete actions by addressing temporal and numeric domains. To deal with time, we formulate invariance conditions that account for both the entire structure of the operators (including the conditions, rather than just the effects) and the possible interactions between operators.

## Introduction

A number of planning domain specification languages used to describe complex real-world planning problems adopt a constraint-based representation centered on multi-valued state variables. Examples of large temporal systems based on such languages are: EUROPA2 (Frank and Jónsson 2003), ASPEN (Chien et al. 2000), IxTeT (Ghallab and Laruelle 1994), HSTS (Muscettola 1994), OMPS (Fratini, Pecora, and Cesta 2008) and Plantrol (Do et al. 2011).

In contrast, the majority of the benchmark domains currently used by the planning community were developed for the International Planning Competitions (IPCs) and are therefore encoded in the PDDL language, which is propositional in nature. Tools designed for translating propositional representations into variable/value representations would facilitate the testing of application-oriented planners on these benchmarks. Designing such tools is primarily concerned with the generation of multi-valued state variables from propositions and operators, which does not depend on the target language of the translation.

This paper presents a technique for generating temporal multi-valued state variables from a PDDL2.2 instance. More specifically, we describe a technique for identifying *temporal mutual exclusion invariants*, which state that certain atoms can never be true at the same time, as a preliminary step to synthesizing state variables. In fact, each identified group of mutually exclusive atoms constitutes the domain of a single state variable.



Our technique builds on the invariant synthesis presented in Helmert 2009 which is used to translate a subset of PDDL2.2 into FDR (Finite Domain Representation), a multi-valued planning task formalism used within the planner Fast Downward (Helmert 2006). Helmert's invariant synthesis is limited to non-temporal and non-numeric PDDL2.2 domains (the so called, PDDL "Level 1"). In contrast, our technique addresses temporal and numeric domains (PDDL – "Level 3"). Developing invariants for such tasks is more complex than handling tasks with instantaneous discrete actions, because interference between concurrent operators complicates the identification of state variables. For this reason, a simple generalization of Helmert's approach does not work in temporal settings. In extending the theory to capture the temporal case, we have had to formulate invariance conditions that take into account the entire structure of the operators (including the conditions, as opposed to the effects only) as well as the possible interactions between them. As a result, we have constructed a significantly more comprehensive technique that is able to find not only invariants for temporal domains, but also a broader set of invariants for non-temporal domains.

This paper is organized as follows. We first identify a set of initial invariant candidates by inspecting the domain. We then check these candidates against a set of properties that assure invariance. If a candidate turns out not to be an invariant, we show that in some cases it is possible to refine it so as to make it a real invariant. An experimental evaluation of our approach and a presentation of conclusions and future work close the paper.

## Invariant Candidates

An ***invariant*** is a property of world states such that when it is satisfied by a state $s$, it is satisfied by all states that are reachable from $s$. Usually, we are interested in invariants that are satisfied in the initial state. If an invariant holds in the initial state, it holds in all the reachable states. Here, we focus on ***mutual exclusion*** invariants, which state that certain atoms can never be true at the same time. For example, if we take the *Logistics* domain, a mutual exclusion invariant for this domain states that two atoms indicating the position of a truck `trk0`, such as `at(trk0, loc0)` and `at(trk0, loc1)`, can never be true at the same time. Intuitively, this means that the truck cannot be at two different

positions simultaneously.

More formally, let $\mathrm{I} = (\mathcal{D}, \mathcal{P})$ be a PDDL instance, where $\mathcal{D}$ is a planning domain and $\mathcal{P}$ a planning problem, an ***invariant candidate*** is a tuple $\mathcal{C} = \langle \Phi, \mathrm{F}, \mathrm{V} \rangle$, where $\Phi$ is a non-empty subset of the atoms in the domain $\mathcal{D}$, and F and V are two disjoint sets of variables. The atoms in $\Phi$ are called the candidate's *components*, while the two sets F and V are respectively called *fixed* and *counted* variables. They are both subsets of $\mathrm{Var}[\Phi]$, which collects the variables in $\Phi$. For example, if we take the *Logistics* domain and the predicate `at(truck,loc)`, the following is a candidate: $\mathcal{C}_{at} = \langle \{$`at(truck,loc)`$\}, \{$`truck`$\}, \{$`loc`$\} \rangle$, where `at(truck,loc)` is the only component of this candidate, `truck` is the fixed variable and `loc` the counted variable.

An ***instance*** $\gamma$ of the candidate $\mathcal{C}$ is a function that maps the fixed variables in F to objects of the problem $\mathcal{P}$. Assuming we have a problem with two trucks `trk1` and `trk2`, we have two possible instances of $\mathcal{C}_{at}$: $\gamma_{trk1} :$ `truck` $\rightarrow$ `trk1` and $\gamma_{trk2} :$ `truck` $\rightarrow$ `trk2`.

The ***weight*** of an instance $\gamma$ in a state $s$ is the number of ground instantiations of the variables in V that make some $\phi \in \Phi$ true under $\gamma$ in $s$. Thus, considering the *Logistics* domain and the instance $\gamma_{trk1}$, if we have a state $s$ where the atom `at(trk1,loc1)` holds, then the weight of $\mathcal{C}_{at}$ is one. Intuitively, the weight of $\gamma$ in a state $s$ is the number of the candidate's components that are true in $s$ when the fixed variables have been instantiated according to $\gamma$.

Given a ***cardinality set*** $\mathrm{S} = \{x \mid x \in \mathbb{N}\}$, the semantics of a candidate $\mathcal{C}$ is: for all the possible instances $\gamma$ of $\mathcal{C}$, if the weight of $\gamma$ is within S in a state $s$, then it is within S in any successor state $s'$ of $s$. Thus, if we prove that the candidate $\mathcal{C}$ holds (i.e. $\mathcal{C}$ is an invariant) and is satisfied in the initial state, we have that at most $k = \max(\mathrm{S})$ atoms in $\Phi$ are true in any reachable state. Since we focus on finding mutually exclusive sets of propositions, we are interested in cases in which at most one atom in $\Phi$ is true in any reachable state. Considering the *Logistics* domain again, the candidate $\mathcal{C}_{at}$ means that, for each truck `trk` in the domain, if the number of locations `loc` where `at(trk,loc)` is true is at most one in a state $s$, then it is at most one in any successor state $s'$ of $s$. If we prove that what is stated by the candidate is true and each truck is at a maximum of one location in the initial state, then each truck cannot be at multiple locations at the same time in any reachable state. Hence, for each truck, we can create a state variable that corresponds to the predicate `at` and represents the position of the truck. The values of this variable represent the presence of the truck in the various locations that it can occupy.

In Helmert's work, he considers only the cardinality set $\mathrm{S} = \{1\}$. However, we consider the set $\mathrm{S} = \{0, 1\}$ because, with durative actions, it is common for a proposition to be deleted at the beginning of an action (e.g. the location of an object being moved), and replaced by a new proposition at the end of the action (e.g. the new location of the object). This corresponds to a decrease in the weight of $\gamma$ to zero at the beginning of the action, and an increase back to one at the end. Allowing $\mathrm{S} = \{0, 1\}$ could be useful in non-temporal domains as well, since it allows operators bringing the weight from zero to one to be classified as safe for invariance conditions. This approach therefore allows us to find more invariants than the techniques using only $\mathrm{S} = \{1\}$. Although we focus here on $\mathrm{S} = \{0, 1\}$, our technique for finding invariants can be generalized to larger cardinality sets.

## Invariance Conditions

In order to show that a candidate $\mathcal{C}$ is an actual invariant, we need to guarantee that, for any instance $\gamma$ of $\mathcal{C}$, the weight of $\gamma$ is within the cardinality set $\mathrm{S} = \{0, 1\}$ in the initial state and all the operators in the domain $\mathcal{D}$ keep the weight within this set. When an operator satisfies this condition, we say that it is *safe* and so it does not *threaten* the candidate $\mathcal{C}$.

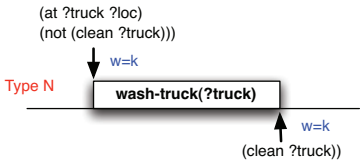
More formally, given an instance $\gamma$ of a candidate $\mathcal{C}$, an operator *op* is ***safe*** if, for any situation where: i) the weight of $\gamma$ is less than or equal to one prior to executing *op* and ii) it is legal to execute *op*, the weight of $\gamma$ is guaranteed to remain less than or equal to one through the execution of *op* and immediately following *op*. A domain $\mathcal{D}$ is safe for $\mathcal{C}$ if and only if all operators in $\mathcal{D}$ are safe for any instance $\gamma$ of $\mathcal{C}$.

A *sufficient condition* for $\mathcal{C}$ to be an actual invariant is that the domain is safe for $\mathcal{C}$.

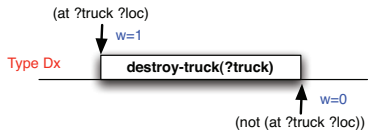
Given a candidate $\mathcal{C}$ and an instance $\gamma$, when can we ensure that an operator *op* is safe, i.e. maintains the weight of $\gamma$ within the cardinality set $\mathrm{S} = \{0, 1\}$? Clearly, if the operator does not change the weight of $\gamma$, then it is safe. On the other hand, if an operator increases the weight of $\gamma$ by two or more at any time-point, it is definitely not safe. If the operator increases the weight of $\gamma$ by one, there might be circumstances in which it is safe, depending on the structure of the conditions and the effects of the operator itself and on its interactions with other operators.

Given an instance $\gamma$ of a candidate $\mathcal{C}$, an operator *op* is safe if it falls in one of the following six categories:

1. **Type N - Inert.** The operator *op* does not affect the weight of $\gamma$. Clearly, an inert operator is safe because it preserves the weight of $\gamma$. Considering a simple *Logistics* domain, the figure below shows an example of such an operator with respect to the candidate $\mathcal{C} = \langle \{$`at(truck,loc)`$\}, \{$`truck`$\}, \{$`loc`$\} \rangle$.

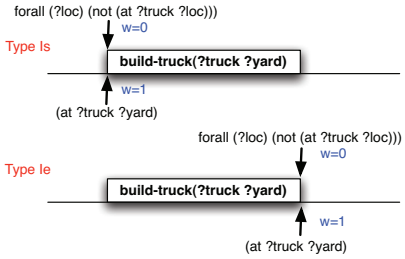



2. **Type D: Decreasing.** The operator *op* decreases the weight of $\gamma$ at some time-point, and does not increase it at any time point. A decreasing operator may or may not have a condition on $\gamma$, and the decrease may even be universally quantified. Like an inert operator, a decreasing operator is safe because it does not cause an increase in the weight at any time-point, and therefore maintains the weight within the cardinality set $\mathrm{S} = \{0, 1\}$. The figure below shows one of several possible decreasing operators with respect to the candidate $\mathcal{C} = \langle \{$`at(truck,loc)`$\}, \{$`truck`$\}, \{$`loc`$\} \rangle$.

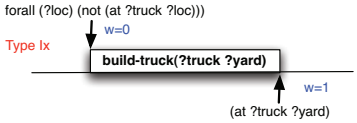



3. **Type I: Increasing.** The operator *op* increases the weight of $\gamma$ from zero to one. We identify three possible subcases:

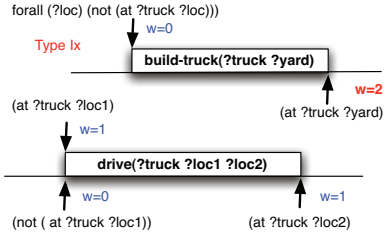
   - **Type Is and Ie**: The operator *op* increases the weight of $\gamma$ by one at some time-point (start/end) and its conditions require that the weight of $\gamma$ be zero at the same time-point (start/end). Increasing operators of type Is and Ie are safe because they bring the weight from zero to one at just one time-point. The figure below shows an increasing operator at start and an increasing operator at end with respect to the candidate $\mathcal{C} = \langle\{\texttt{at(truck,loc)}\}, \{\texttt{truck}\}, \{\texttt{loc}\}\rangle$.



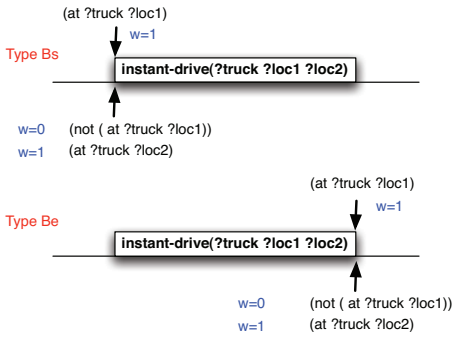

   - **Type Ix**: a condition at start guarantees that the weight of $\gamma$ is zero and an add effect at end increases the weight by one. The figure below shows an example of such an operator with respect to the candidate $\mathcal{C} = \langle\{\texttt{at(truck,loc)}\}, \{\texttt{truck}\}, \{\texttt{loc}\}\rangle$.




     An operator of type Ix is safe if it is mutex with all those operators that may increase the weight of $\gamma$ over its duration. The following picture shows a simple example of when this might happen.



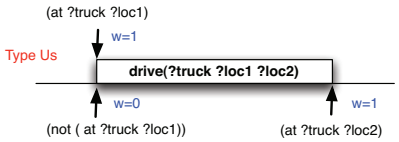

4. **Type B: Balanced.** The operator *op* preserves the weight of $\gamma$ by checking that the weight is one at some time-point (start/end), decreasing the weight by one at that time-point and then bringing back the weight to one at that same time-point. Balanced operators are always safe because they act at only one time-point (start/end) and do not change the overall weight of $\gamma$. The figure below shows a balanced operator at start (**Type Bs**) and a balanced operator at end (**Type Be**) with respect to the candidate $\mathcal{C} = \langle\{\texttt{at(truck,loc)}\}, \{\texttt{truck}\}, \{\texttt{loc}\}\rangle$.



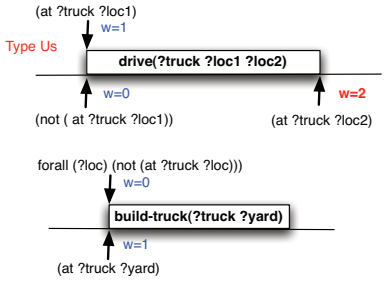

5. **Type U: Temporarily Unbalanced.** The operator *op* ensures that the weight of $\gamma$ is one at start, brings the weight from one to zero at start or at end and then restores the weight to one at end.

   We have two different configurations for a temporarily unbalanced operator:

   - **Type Us**: a condition at start guarantees that the weight is one, a delete effect at start decreases the weight from one to zero, and an add effect at end restores the weight to one. The figure below shows an example of such an operator with respect to the candidate $\mathcal{C} = \langle\{\texttt{at(truck,loc)}\}, \{\texttt{truck}\}, \{\texttt{loc}\}\rangle$.

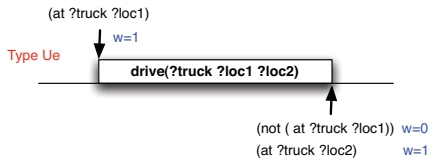



     An unbalanced operator of type Us is safe if it is mutex with all those operators that may increase the weight of $\gamma$ over its duration. The following picture shows a simple example of when this might happen.

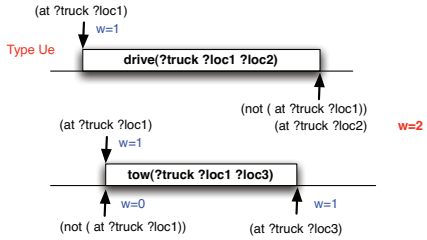



     Unbalanced operators of type Us are particularly common because they model the usage of renewable resources. A renewable resource is needed during the execution of the action, so the weight goes from one to zero at start, but it is not consumed by the action, so the weight returns to one at end.

   - **Type Ue**: a condition at start guarantees that the weight is one and a delete and an add effect at end bring the weight from one to zero and then back to one. The figure below shows an example of such an operator with respect to the candidate $\mathcal{C} = \langle\{\texttt{at(truck,loc)}\}, \{\texttt{truck}\}, \{\texttt{loc}\}\rangle$.



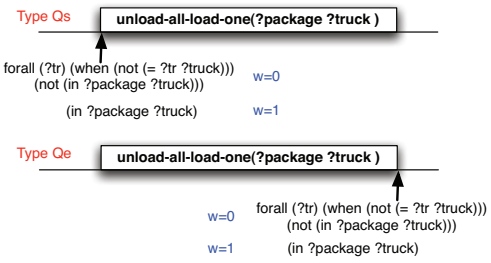

An unbalanced operator of type Ue is safe if it is mutex with all operators that may alter the weight during its execution. Although this operator does not cause an overall change in the weight of $\gamma$ when executed in isolation, it might give rise to problematic situations when another operator $op_i$ capable of changing the weight is allowed to take place over its duration. This is because the application of $op_i$ may have the side effect of making the delete effect of $op$ no longer applicable, which would in turn provoke an overall increase of the weight by two instead of one. The figure below exemplifies this situation.



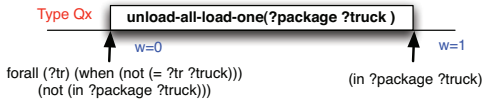

One could argue that unbalanced operators of type Ue originate from a faulty description of renewable resources and so they should in reality be operators of type Us. We have found a few examples of operators of type Ue in the domains of previous IPCs, but none resulted in provable invariants.

6. **Type Q: Quantified Delete.** The operator $op$ sets the weight of $\gamma$ to zero at some time-point (start/end) through a universally quantified delete effect and then brings back the weight to one at the same time-point (start/end) or after that. We distinguish three possible sub-cases:

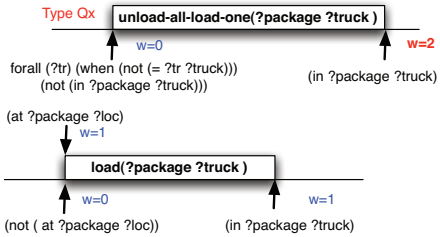
   - **Type Qs and Qe**: a universally quantified effect sets the weight to zero at some time-point (start/end) and an add effect increases the weight by one at the same time-point (start/end). Operators of type Qs and Qe are safe because they ensure that only the single add effect will be true. The figure below shows an example of such operators with respect to the candidate $\mathcal{C} = \langle\{\texttt{in(package,truck)}\}, \{\texttt{package}\}, \{\texttt{truck}\}\rangle$.




   - **Type Qx**: a universally quantified effect at start sets the weight to zero and an add effect at end increases the weight by one. The figure below shows an example of such an operator with respect to the candidate $\mathcal{C} = \langle\{\texttt{in(package,truck)}\}, \{\texttt{package}\}, \{\texttt{truck}\}\rangle$.




An unbalanced operator of type Qx is safe if it is mutex with all those operators that may alter the weight during its execution. The following picture shows an example of when this might happen.




Inert and balanced safe operators represent the temporal generalization of the non-threatening operators used in Helmert's invariant synthesis (Helmert 2009). The criteria for identifying increasing, decreasing and quantified delete operators can be readapted for use in non-temporal planning domains. They correspond to the use of the cardinality set $\mathrm{S} = \{0, 1\}$ instead of $\mathrm{S} = \{1\}$, which allows us to capture a broader set of invariants than Helmert's approach. In contrast, unbalanced operators are specific to temporal planning and correspond to cases where the effects of an action are not fully realized until the end. Such operators can still be safe, as long as no other operator can disrupt the candidate during the execution of the operator.

## Temporal Mutex Conditions

We now clarify the exact nature of the temporal mutex conditions that must hold in order to ensure the safeness of unbalanced operators and operators whose effects are split over time, such as operators of type Dx, Ix, and Qx.

In order to assess if an operator $op$ is safe, we first need to establish what kinds of operators may disrupt the weight during the execution of $op$ and then specify the exact mutex relationships that must hold between $op$ and the possibly disrupting operators.

Let us consider the second issue first. In general, how can we establish whether two durative PDDL operators are mutex or not? Since in PDDL2.2, effects can only happen at the start and end of the operators, and conditions can only be specified at the start, end, and over all, there are nine types of mutex. We refer the reader to (Smith and Jónsson 2002, ) for a discussion of mutex between actions with general conditions and effects.

Given two durative operators $op_1$ and $op_2$, these nine types of ***mutex operators*** are the following:

1. ***Start-Start***: $op_1$ and $op_2$ cannot start at the same time if:
   $\exists p \in (\mathrm{Cond}_{start}(op1) \cup \mathrm{Eff}_{start}(op1)) :$
   $\neg p \in (\mathrm{Cond}_{start}(op2) \cup \mathrm{Eff}_{start}(op2))$

2. ***End-End***: $op_1$ and $op_2$ cannot end at the same time if:
   $\exists p \in (\mathrm{Cond}_{end}(op1) \cup \mathrm{Eff}_{end}(op1)) :$
   $\neg p \in (\mathrm{Cond}_{end}(op2) \cup \mathrm{Eff}_{end}(op2))$
3. ***Start-End***: $op_1$ cannot start at the time that $op_2$ ends if:
   $\exists p \in (\mathrm{Cond}_{start}(op1) \cup \mathrm{Eff}_{start}(op1)) :$
   $\neg p \in (\mathrm{Cond}_{end}(op2) \cup \mathrm{Eff}_{end}(op2))$
4. ***Invariant-Start***: $op_2$ cannot start during $op_1$ if:
   $\exists p \in \mathrm{Cond}_{all}(op1) :$
   $\neg p \in (\mathrm{Cond}_{start}(op2) \cup \mathrm{Eff}_{start}(op2))$
5. ***Invariant-End***: $op_2$ cannot end during $op_1$ if:
   $\exists p \in \mathrm{Cond}_{all}(op1) :$
   $\neg p \in (\mathrm{Cond}_{end}(op2) \cup \mathrm{Eff}_{end}(op2))$
6. ***Invariant-Invariant***: $op_1$ and $op_2$ cannot overlap if:
   $\exists p \in \mathrm{Cond}_{all}(op1) : \neg p \in \mathrm{Cond}_{all}(op2)$

In addition, we have: 7. mutex ***End-Start*** (dual to case 3), 8. mutex ***Start-Invariant*** (dual to case 4) and 9. mutex ***End-Invariant*** (dual to case 5). For brevity, we refer to such mutexes as *mutex-SS*, *mutex-EE*, and so on.

As for identifying possibly disrupting operators, we need to reason about the operators in the domain according to two criteria: i) what type of legal weight change they produce (from zero to one, from one to zero or from one to zero to one); and ii) at what time-points the changes happen.

Following this reasoning, for each type of unbalanced operator *op*, we identify a set of ***mutex constraints*** that involve *op* and those operators that can possibly disrupt its weight. If these constraints are satisfied, then *op* is safe.

- An operator of type Ix or Qx is safe if it is:
  1. mutex SS and IS and ES with operators of type (I,Q)s
  2. mutex SE and IE and EE with operators of type Us, (I,Q)x, and (I,Q,U)e
- An operator of type Us is safe if it is:
  1. mutex SS and IS and ES with operator of type Qs
  2. mutex IS and ES with operator of type Is
  3. mutex SE and IE and EE with operators of type Ue, (I,Q)x, and Qe
  4. mutex IE and EE with operators of type Ie
- An operator of type Ue is safe if it is:
  1. mutex SS and IS and ES with operators of type (Q,B)s
  2. mutex IS and ES with operators of type Is
  3. mutex SE and IE and EE with operators of type Us, (I,Q)x, and (Q,B,U)e
  4. mutex IE and EE with operators of type Ie

### Decision Tree

In Figure 1, we show a binary ***decision tree*** $\mathcal{T}$ that can be used to determine whether an operator *op* is safe w.r.t an instance $\gamma$ of a candidate $\mathcal{C}$ or not. The internal nodes of the tree test the structure of the conditions and effects of the operator. The abbreviations stand for: Add-s $\rightarrow$ add effect at start, Del-s $\rightarrow$ delete effect at start, W=0-s $\rightarrow$ weight is zero at start, UQ del-s $\rightarrow$ universally quantified delete effect at start. Abbreviations for conditions and effects at end are analogous. On the basis of the configuration of the conditions and effects of the operator *op*, the leaf nodes assign a classification: either *op* is safe or it is unsafe (respectively, "OK" and "X" in the tree). The leaves of the tree marked with "OK" represent all the possible cases in which we accept an operator as safe. Green labels in the figure link these cases with the six categories of safe operators described above. Close to the corresponding branches of the tree, we also give a graphical representation of the configuration of the operator's conditions and effects. It is worth noting that a few of the operators in the tree are quite bizarre and unlikely to appear in practice. For example, operators of type 1 (such as IsIe) could not even be executed without required concurrency – some other operator would have to reduce the weight back to zero in the middle. Nevertheless, we have included these operators in the tree for completeness.

## Guess, Check and Repair Algorithm

As with other related techniques (Helmert 2009), our algorithm for finding invariants implements a *guess, check and repair* approach. We start from a simple set of initial candidates and use the decision tree in Figure 1 to evaluate if each candidate $\mathcal{C}$ is an invariant. If we reach a failure leaf for any operator *op* in the domain, before discarding $\mathcal{C}$, we identify what features of *op* threaten $\mathcal{C}$ and exploit this knowledge for creating new candidates that are guaranteed not to be threatened by the same operator *op*. These new candidates need to be checked against the invariance conditions and might fail due to different threatening operators. The tree in Figure 1 associates, whenever possible, a set of fixes to dead leaves.

When we create the set of initial candidates, we ignore constant predicates, i.e. predicates whose atoms have the same truth value in all the states (for example, type predicates). In fact, they are trivially invariants and so are typically not interesting. Among the modifiable atoms, we use initial predicates with the following characteristics: the set $\Phi$ contains only one atom $\phi$ and the set V contains only one counted variable. The candidate $\mathcal{C}_{at} = \langle\{\texttt{at(truck,loc)}\}, \{\texttt{truck}\}, \{\texttt{loc}\}\rangle$ is an example of an initial candidate. This choice comes from experience and is the same as for other related techniques (Helmert 2009).

Given an initial candidate, we test the safety of each operator in the domain by traversing the decision tree in Figure 1. The main difficulty associated with traversing the tree is that we can check the mutex constraints associated with some branches of the tree only when we know the type of each operator. The simplest way to handle this is to make two iterations: the first to classify operators according to types and the second to check the operators. However, we follow a more efficient approach by checking most of the operators during the first iteration, and just returning to do the mutex checks for those operators that require them, after all of the operators have been classified. We apply the following procedure:

1. Select a candidate invariant $\mathcal{C}$ and traverse the decision tree $\mathcal{T}$ for each operator in the domain $\mathcal{D}$.

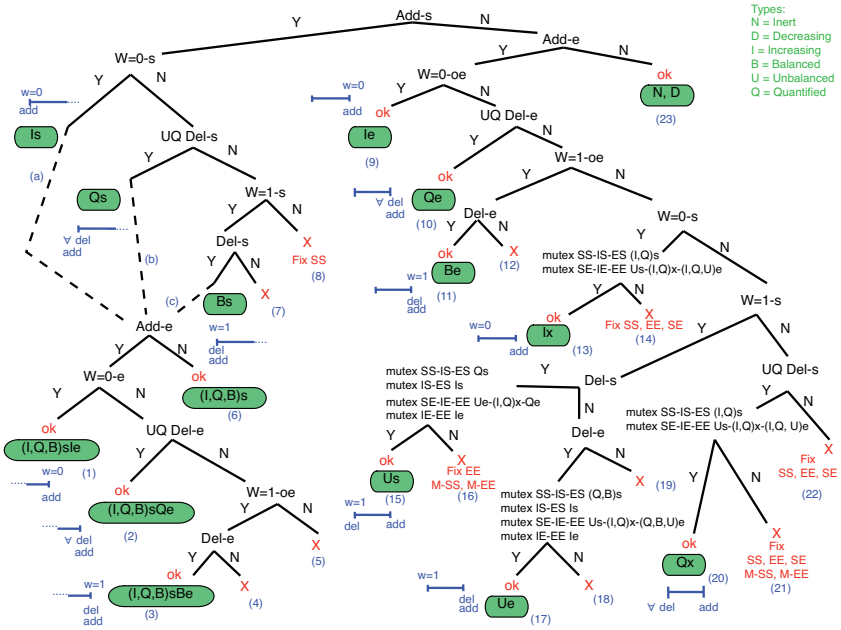



Figure 1: Decision Tree $\mathcal{T}$ for checking whether an operator *op* is safe w.r.t an instance $\gamma$ of a candidate $\mathcal{C}$.

2. If a node requiring a mutex check is reached for an operator *op*, save *op* in a bucket for that mutex check and proceed as if the mutex check succeeded.
3. Run the corresponding mutex checks for the operators in the buckets.
4. At any point in the process, if a failure leaf node is reached, discard the candidate $\mathcal{C}$.
5. At any point in the process, if a fix leaf node is reached, generate a new candidate for each possible fix, and start the process over.

Step 2 in the above procedure classifies the operators according to the six types described in the previous section.

## Refining Candidates

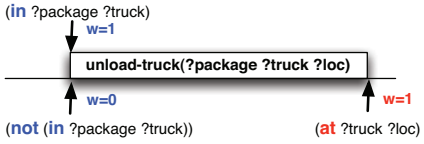
The choice of how to fix a failed candidate depends on the features of the operators that threaten it. More specifically, given a candidate $\mathcal{C} = \langle \Phi, \mathrm{F}, \mathrm{V} \rangle$ that has been rejected because it is threatened by an operator *op*, we refine $\mathcal{C}$ by picking a new atom $\phi$, which is chosen on the basis of the structure of *op* as explained below, and adding it to the components' set $\Phi$ of $\mathcal{C}$. So, we obtain the new candidate $\mathcal{C}' = \langle \{\Phi \cup \phi\}, \mathrm{F}', \mathrm{V}' \rangle$, which will not fail for the same reasons as $\mathcal{C}$, but might fail for different reasons. The new atom $\phi$ must involve only the variables in F and at most one other variable and must satisfy one of the following three criteria:

1. ***Fix SS:*** the atom $\phi$ unifies with a positive condition at start and a delete effect at start of *op*.
2. ***Fix EE:*** the atom $\phi$ unifies with a positive condition at end (or over all) and a delete effect at end of *op*.
3. ***Fix SE:*** the atom $\phi$ unifies with a positive condition at start and a delete effect at end of *op*.

Given a candidate $\mathcal{C}$ and an instance $\gamma$, we apply fixes SS, EE and SE in the following cases:

1. Fix SS when $\mathcal{C}$ is threatened by an operator *op* such that:
   - *op* has an add effect at start or at end increasing the weight of $\gamma$, but no delete effects or conditions involving $\gamma$ either at start or at end (respectively, Leaf 8 and Leaf 22 in the decision tree in Figure 1);
   - *op* has the same configuration as safe operators of type Ix or Qx, but it is actually unsafe because it does not satisfy the mutex conditions that ensure the weight remains within the cardinality set $\mathrm{S} = \{0, 1\}$ during its execution (respectively, Leaf 14 and Leaf 21).
2. Fix EE when $\mathcal{C}$ is threatened by an operator *op* such that:
   - *op* is of type 14, 21 and 22 just described above;
   - *op* has the same configuration as safe operators of type Us, but it is actually unsafe because it does not satisfy the mutex conditions that ensure the weight remains within the cardinality set $\mathrm{S} = \{0, 1\}$ during its execution (Leaf 16);
3. Fix SE when $\mathcal{C}$ is threatened by an operator *op* of type 14, 21 and 22 just described above.

As an example, consider the *Logistics* domain with the operator `unload-truck`, shown in the figure below. For the




candidate $\mathcal{C}_{at} = \langle \{\texttt{at(package,loc)}\}, \{\texttt{package}\}, \{\texttt{loc}\} \rangle$, we see that operator `unload-truck` threatens

$\mathcal{C}_{at}$ because it increases the weight at end without decreasing it or checking that the weight is zero. If we traverse the tree in Figure 1 guided by the conditions and effects of the operator `unload-truck`, we reach leaf 22. Although this is a failure leaf, it indicates that, before discarding $\mathcal{C}_{at}$, we can try to apply fixes SS, EE and SE. Fix SS can be used in this case because the atom $\phi$ =`in(?package ?truck)` appears both in the positive conditions at start and in the delete effects at start. Therefore, we add the candidate $\mathcal{C}_{at/in}$ = $\langle$`{at(package,loc), in(package,truck)}`, `{package}`, `{loc, truck}`$\rangle$ to the list of candidates to check. By evaluating the new candidate $\mathcal{C}_{at/in}$ against the invariance conditions, we will conclude that $\mathcal{C}_{at/in}$ is in fact an invariant.

## Experimental Results

In this section, we present some experimental results for the invariant synthesis technique developed above. The current version of the algorithm is implemented in the Python language. The experiments were conducted by using a 2.53 GHz Intel Core 2 Duo processor with a memory of 4 GB.

Below, we present the invariants that the algorithm finds for some temporal domains of the IPC-2008. Each invariant is enclosed in braces where the predicate names indicate the components of the invariant, numbers not enclosed in square brackets indicate the position of the fixed variables in the list of arguments of the corresponding predicate and numbers enclosed in square brackets indicate the counted variables. For example, considering our running example, {at 0 [1], in 0 [1]} indicates the invariant having –at(package, location)in(package,vehicle)˝ as components, package as a fixed variable, and –location,vehicle˝ as counted variables.

- *Elevators-strips:*
  `{passengers 0 [1]}`
  `{lift-at 0 [1]}`
  `{passenger-at 0 [1], boarded 0 [1]}`
- *Sokoban-strips:*
  `{at 0 [1]}`
  `{at 1 [0], clear 0}`
  `{clear [0]}`

Table 1 compares the number of invariants (# INV) found by the Temporal Invariant Synthesis (TIS) just discussed with those found by a Simple version of the Invariant Synthesis (SIS) for the temporal domains of the IPC-6, IPC-5, IPC-4 and IPC-3. The SIS represents a simple generalization of Helmert's invariant synthesis (Helmert 2009) to the temporal case.

Table 1 also reports the number of invariants obtained by applying fixes (# FIX) and run time (RT) for generating invariants for the temporal domains. The computational time is negligible; there is no significant delay associated with either checking a broad set of configurations in the operators' conditions and effects or performing the mutex checks.

For the invariants found by our algorithm, the most common operators are of type 23, which means that the operator does not even potentially threaten the invariant because it is inert or decreasing, and type 15, which corresponds to the usage of a renewable resource. We also found operators of types 6c, 11, 15, and 23. Additional operators of types 8, 12, 16, 17, 18, and 22 were found while examining invariant candidates that were ultimately rejected. Based on discussions with Will Cushing, these findings appear to be consistent with his analysis (Cushing et al. 2007).

| Domains | # INV SIS | # INV TIS | # FIX TIS | RT TIS |
|---|---|---|---|---|
| Crew Planning-IPC-6 | 0 | 3 | 0 | 0.0054 |
| Elevators-Num-IPC-6 | 0 | 2 | 1 | 0.0025 |
| Elevators-Str-IPC-6 | 0 | 3 | 1 | 0.0037 |
| Modeltrain-Num-IPC-6 | 3 | 7 | 1 | 0.0089 |
| Openstacks-Adl-IPC-6 | 2 | 7 | 4 | 0.0043 |
| Openstacks-Num-IPC-6 | 4 | 10 | 6 | 0.0054 |
| Openstacks-Num-Adl-IPC-6 | 2 | 6 | 4 | 0.0030 |
| Openstacks-Str-IPC-6 | 4 | 11 | 6 | 0.0073 |
| Parcprinter-Str-IPC-6 | 5 | 7 | 2 | 0.0126 |
| Pegsol-Str-IPC-6 | 0 | 2 | 1 | 0.0008 |
| Sokoban-Str-IPC-6 | 0 | 3 | 1 | 0.0033 |
| Transport-Num-IPC-6 | 0 | 3 | 1 | 0.0030 |
| Woodworking-Num-IPC-6 | 2 | 7 | 3 | 0.0167 |
| Openstacks-IPC-5 | 2 | 7 | 4 | 0.0048 |
| Pathways-IPC-5 | 0 | 0 | 0 | 0.0003 |
| Pipesworld-IPC-5 | 0 | 8 | 7 | 0.0266 |
| Rovers-IPC-5 | 4 | 9 | 0 | 0.0142 |
| Storage-IPC-5 | 0 | 3 | 2 | 0.0071 |
| TPP-IPC-5 | 0 | 1 | 0 | 0.0006 |
| Trucks-IPC-5 | 0 | 2 | 2 | 0.0055 |
| Airport-IPC-4 | 2 | 2 | 0 | 0.0399 |
| Pipesworld-NT-IPC-4 | 0 | 4 | 4 | 0.0162 |
| Pipesworld-T-IPC-4 | 0 | 8 | 7 | 0.0270 |
| Satellite-IPC-4 | 0 | 2 | 1 | 0.0027 |
| UMTS-4 | 0 | 0 | 0 | 0.0079 |
| Depots-IPC-3 | 0 | 6 | 5 | 0.0113 |
| DriverLog-IPC-3 | 0 | 2 | 2 | 0.0051 |
| ZenoTravel-IPC-3 | 0 | 1 | 1 | 0.0031 |
| Rovers-IPC-3 | 4 | 9 | 0 | 0.0137 |
| Satellite-IPC-3 | 0 | 2 | 1 | 0.0027 |

Table 1: Number of invariants (# INV), number of invariants coming from fixes (# FIX) and run time (RT) for generating invariants for the temporal domains of the IPCs by using the Temporal Invariant Synthesis (TIS) and the Simple Invariant Synthesis (SIS).

Table 2 shows a comparison between the number of state variables obtained by instantiating invariants for domains of the IPC-6 coming from a Naive Invariant Synthesis (NIS), which basically produces a state variable with two truth values (true and false) for each atom in the domain, the Simple Invariant Synthesis (SIS), and our Temporal Invariant Synthesis (TIS). In many domains the TIS yields a significant reduction in the number of state variables in comparison with the other two techniques. In six instances of Elevators-str, Sokoban-str, and Transport-Num the reduction is greater than an order of magnitude.

## Conclusions and Future Work

In this paper, we presented a technique for automatically synthesizing invariants starting from temporal planning domains expressed in PDDL2.2. Our technique builds on Helmert's invariant synthesis (Helmert 2009), but extends it to apply to temporal domains and also identifies a broader set of invariants. This is achieved by considering the cardinality set $\mathrm{S} = \{0, 1\}$ instead of $\mathrm{S} = \{1\}$ and by analyzing

| Domains | # SV | | |
|---|---|---|---|
| | NIS | SIS | TIS |
| Crew Planning - p10 | 112 | 112 | 106 |
| Crew Planning - p20 | 305 | 305 | 261 |
| Crew Planning - p30 | 510 | 510 | 498 |
| Elevators-Str - p10 | 203 | 203 | 21 |
| Elevators-Str - p20 | 592 | 592 | 34 |
| Elevators-Str - p30 | 1240 | 1240 | 49 |
| Openstacks-Num - p10 | 71 | 71 | 29 |
| Openstacks-Num - p20 | 121 | 121 | 49 |
| Openstacks-Num - p30 | 171 | 171 | 69 |
| Modeltrain-Num - p10 | 397 | 205 | 191 |
| Modeltrain-Num - p20 | 396 | 204 | 188 |
| Modeltrain-Num - p30 | 910 | 418 | 390 |
| Parcprinter-Str - p10 | 641 | 641 | 431 |
| Parcprinter-Str - p20 | 1273 | 1273 | 673 |
| Parcprinter-Str - p30 | 669 | 669 | 439 |
| Pegsol-Str - p10 | 66 | 66 | 33 |
| Pegsol-Str - p20 | 66 | 66 | 33 |
| Pegsol-Str - p30 | 66 | 66 | 33 |
| Sokoban-Str - p10 | 490 | 490 | 72 |
| Sokoban-Str - p20 | 127 | 127 | 37 |
| Sokoban-Str - p30 | 1131 | 1131 | 75 |
| Transport-Num - p10 | 1292 | 1292 | 36 |
| Transport-Num - p20 | 1292 | 1292 | 36 |
| Transport-Num - p30 | 1772 | 1772 | 64 |
| Woodworking-Num - p10 | 143 | 143 | 95 |
| Woodworking-Num - p20 | 239 | 239 | 151 |
| Woodworking-Num - p30 | 251 | 251 | 158 |

Table 2: Number of state variables (# SV) for temporal domains of the IPC-6 that are obtained by instantiating invariants coming from: (1) a Naive Invariant Synthesis (NIS); (2) a Simple Invariant Synthesis (SIS); and (3) our Temporal Invariant Synthesis (TIS).

the entire structure of an operator to assess its safety with respect to an invariant. Finding a wider set of invariants allows us to synthesize a smaller number of state variables to represent a domain. All the temporal planners that use state variables to represent the world greatly benefit from dealing with a relatively small number of state variables.

Our technique can be incorporated in any translation from PDDL2.2 to a language based on multi-valued state variables. In particular, we have used the temporal invariant synthesis described here in our translator from PDDL2.2 to NDDL, EUROPA2's domain specification language (Bernardini and Smith 2008). The use of this translator, which includes the temporal invariant synthesis described here as one of its core steps, has facilitated the testing of EUROPA2 against domains of the IPCs originally expressed in PDDL2.2.

In the future, we intend to use information about *types*, which are available in PDDL2.2 domains, for identifying a more comprehensive set of invariants. As an example, let us consider a domain in which we have a predicate `(pred ?arg1 - supertype ?arg2 - type)` and the types `subtype1` and `subtype2` are both of type `supertype`. Given an invariant candidate $\mathcal{C} = \langle\{$`pred(arg1,arg2)`$\}, \{$`arg1 - supertype`$\}, \{$`arg2 - type`$\}\rangle$, suppose that no operator threatens $\mathcal{C}$ when `arg1` is bound to an object of type `subtype1`, but an operator *op* threatens $\mathcal{C}$ when `arg1` is bound to an object of type `subtype2`. In this case, our algorithm rejects the candidate and, if no fix involving `pred` can be applied, the algorithm encodes `pred` with binary state variables. However, if we enrich the algorithm with the ability to use information about types, it will consider two more specific candidates $\mathcal{C}_1 = \langle\{$`pred(arg1,arg2)`$\}, \{$`arg1 - subtype1`$\}, \{$`arg2 - type`$\}\rangle$ and $\mathcal{C}_2 = \langle\{$`pred(arg1,arg2)`$\}, \{$`arg1 - subtype2`$\}, \{$`arg2 - type`$\}\rangle$. Now, the algorithm will accept $\mathcal{C}_1$ as an invariant since it is not threatened by any operator, while it will fail $\mathcal{C}_2$ since *op* threatens it.

## Acknowledgments

We thank Malte Helmert and Gabriele Röger for making their code for translating PDDL into FDR available and William Cushing for helpful discussions about the configurations of temporal operators. We are grateful to the anonymous reviewers for their suggestions on earlier drafts of the paper. This work has been supported by the London Knowledge Lab and the NASA Exploration Systems Program.

## References

Bernardini, S., and Smith, D. E. 2008. Translating PDDL2.2 into a constraint-based variable/value language. In *Proc. of the Workshop on Knowledge Engineering for Planning and Scheduling, 18th International Conference on Automated Planning and Scheduling*.

Chien, S.; Rabideau, G.; Knight, R.; Sherwood, R.; Engelhardt, B.; Mutz, D.; Estlin, T.; B.Smith; Fisher, F.; Barret, T.; Stebbins, G.; and Tran, D. 2000. ASPEN - Automated planning and scheduling for space missions operations. In *6th International Conference on Space Operations*.

Cushing, W.; Weld, D.; Kambhampati, S.; Mausam; and Talamadupula, K. 2007. Evaluating temporal planning domains. In *Proc. of the Seventeenth International Conference on Automated Planning and Scheduling*, 105–112.

Do, M.; Pkajima, K.; Uckun, S.; Hasegawa, F.; Kawano, Y.; Tanaka, K.; Crawford, L.; Zhang, Y.; and Ohashi, A. 2011. Online planning for a material control system for liquid crystal display manufacturing. In *Proc. of the Twenty-First International Conference on Automated Planning and Scheduling*, 50–57.

Frank, J., and Jónsson, A. 2003. Constraint based attribute and interval planning. *Journal of Constraints* 8(4):339–364. Special Issue on Planning.

Fratini, S.; Pecora, F.; and Cesta, A. 2008. Unifying Planning and Scheduling as Timelines in a Component-Based Perspective. *Archives of Control Sciences* 18(2):5–45.

Ghallab, M., and Laruelle, H. 1994. Representation and control in IxTeT, a temporal planner. In *Proc. of the Second International Conference on Artificial Intelligence Planning Systems*, 61–67.

Helmert, M. 2006. The Fast Downward planning system. *Journal of Artificial Intelligence Research* 26:191–246.

Helmert, M. 2009. Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence* 3(17):503–535.

Muscettola, N. 1994. HSTS: Integrating planning and scheduling. In Zweben, M., and Fox, M., eds., *Intelligent Scheduling*. Morgan Kauffmann. 451–469.

Smith, D., and Jónsson, A. 2002. The logic of reachability. In *Proc. of the Sixth International Conference on AI Planning and Scheduling*, 379–387.