

Finding Mutual Exclusion Invariants in Temporal Planning Domains

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. We first identify a set of invariant candidates by inspecting the domain and then check these candidates against properties that assure invariance. If these properties are violated, we show that it is sometimes possible to refine a candidate by adding additional propositions and turn it into a real invariant. 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. As a result, we construct a technique that is not only capable of identifying invariants for temporal domains, but is also able to find a broader set of invariants for non-temporal domains than the previous techniques.

Introduction

A number of planning domain specification languages designed and 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) and OMPS (Fratini, Pecora, and Cesta, 2008).

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 (Edelkamp and Hoffmann, 2004; Fox and Long, 2003).

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.

Other approaches to invariant synthesis are available in the literature (Gerevini and Schubert, 2000; Rintanen, 2000; Fox and Long, 1998). Although similarities with our approach can be found, they are more limited in scope because they deal only with STRIPS domains. These techniques have usually been used in combination with SAT-based planners (Kautz and Selman, 1999) or Graphplan-based planners (Blum and Furst, 1997) for improving their performance.

This paper is organized as follows. In the next section, we identify an initial set of invariant candidates by inspecting the domain. We then explain how to check the candidates against a set of properties that assure invariance in order to verify that our initial candidates are actual invariants. If a candidate turns out not to be an invariant at first, we show that in some cases it is possible to refine the candidate 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 $\text{at}(\text{trk0}, \text{loc0})$ and $\text{at}(\text{trk0}, \text{loc1})$, can never be true at the same time. Intuitively, this means that the truck cannot be at two different positions simultaneously.

In order to find invariants, we use a guess, check and repair approach sorting through hypothetical invariants, which we call *invariant candidates*. We generate invariant candidates as described below and then check them against a set of conditions that ensure invariance. If a hypothetical invariant is not found to be valid, we apply a set of refinements to try to make it a real invariant before rejecting it.

Formally, let $I = (\mathcal{D}, \mathcal{P})$ be a PDDL instance, where \mathcal{D} is a planning domain and \mathcal{P} a planning problem, an *invariant candidate* is a tuple $\mathcal{C} = \langle \Phi, F, V \rangle$, where Φ 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 Φ 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 $\text{Var}[\Phi]$, which collects the variables in Φ . For example, if we take the *Logistics* domain and the predicate $\text{at}(\text{truck}, \text{loc})$, the following is a candidate: $\mathcal{C}_{at} = \langle \{\text{at}(\text{truck}, \text{loc})\}, \{\text{truck}\}, \{\text{loc}\} \rangle$, where $\text{at}(\text{truck}, \text{loc})$ is the only component of this candidate, truck is the fixed variable and loc the counted variable.

An *instance* γ 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} : $\gamma_{\text{trk1}} : \text{truck} \rightarrow \text{trk1}$ and $\gamma_{\text{trk2}} : \text{truck} \rightarrow \text{trk2}$.

The *weight* of an instance γ in a state s is the number of ground instantiations of the variables in V that make some $\phi \in \Phi$ true under γ in s .¹ Thus, considering the *Logistics* domain and the instance γ_{trk1} , if we have a state s where the atom $\text{at}(\text{trk1}, \text{loc1})$ holds, then the weight of \mathcal{C}_{at} is one. Intuitively, the weight of γ 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 γ .

Given a *cardinality set* $S = \{x \mid x \in \mathbb{N}\}$, the semantics of a candidate \mathcal{C} is: for all the possible instances γ of \mathcal{C} , if the weight of γ 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 $k = \max(S)$ atoms in Φ 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 Φ 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 $\text{at}(\text{trk}, \text{loc})$ 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 $S = \{1\}$. However, we consider the set $S = \{0, 1\}$ 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 γ to zero at the beginning of the action, and an increase back to one at the end. Allowing $S = \{0, 1\}$ 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 $S = \{1\}$. We will present relevant examples in the following section. Although we focus here on $S = \{0, 1\}$, 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 γ of \mathcal{C} , the weight of γ is within the cardinality set $S = \{0, 1\}$ 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 γ of a candidate \mathcal{C} , an operator op is *safe* if, for any situation where: i) the weight of γ is less than or equal to one prior to executing op and ii) it is legal to execute op , the weight of γ 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 γ of \mathcal{C} .

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

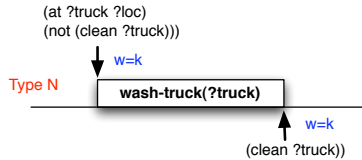
Given a candidate \mathcal{C} and an instance γ , when can we ensure that an operator op is safe, i.e. maintains the weight of γ within the cardinality set $S = \{0, 1\}$? Clearly, if the operator does not change the weight of γ , then it is safe. On the other hand, if an operator increases the weight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

¹The weight of γ is equal to the cardinality of the set of all ground atoms that unify with some $\phi \in \Phi$ under γ in s .

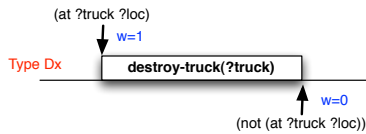
of the conditions and the effects of the operator itself and on its interactions with other operators.

Given an instance γ of a candidate \mathcal{C} , we consider safe an operator op if and only if it falls in one of the following six categories:

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

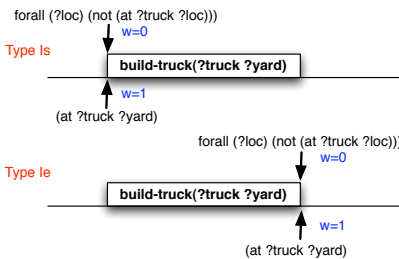


2. **Type D: Decreasing.** The operator op decreases the weight of γ at some time-point, and does not increase it at any time point. A decreasing operator may or may not have a condition on γ , and the decrease may even be universally quantified. Like an inert operator, a decreasing operator is safe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



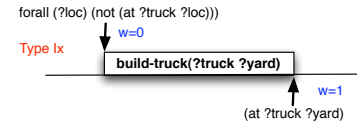
3. **Type I: Increasing.** The operator op increases the weight of γ from zero to one. We identify three possible sub-cases:

- **Type Is and Ie:** The operator op increases the weight of γ by one at some time-point (start/end) and its conditions require that the weight of γ 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 $\mathcal{C} = \langle \{at(truck, loc)\}, \{truck\}, \{loc\} \rangle$.

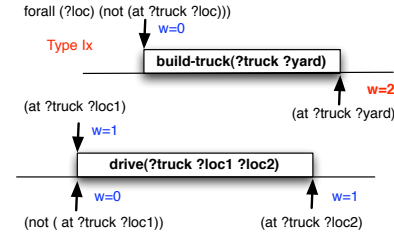


- **Type Ix:** a condition at start guarantees that the weight of γ is zero and an add effect at end increases the

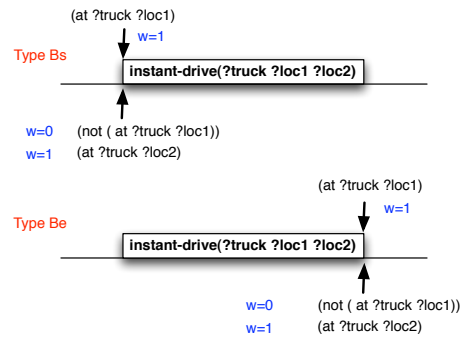
weight by one. The figure below shows an example of such an operator with respect to the candidate $\mathcal{C} = \langle \{at(truck, loc)\}, \{truck\}, \{loc\} \rangle$.



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



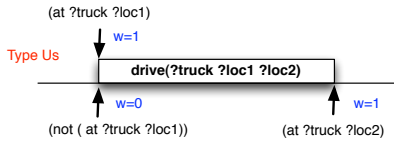
4. **Type B: Balanced.** The operator op preserves the weight of γ by checking that the weight is one at some time-point (start/end), decreasing the weight by one at that time-point and then bringing back the weight to one at that same time-point. Balanced operators are always safe because they act at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



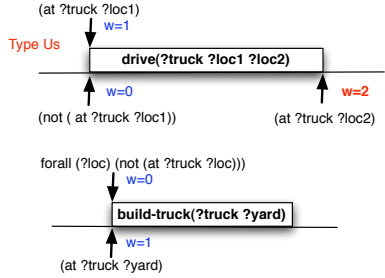
5. **Type U: Temporarily Unbalanced.** The operator op ensures that the weight of γ is one at start, brings the weight from one to zero at start or at end and then restores the weight to one at end.

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 $\mathcal{C} = \langle \{at(truck, loc)\}, \{truck\}, \{loc\} \rangle$.

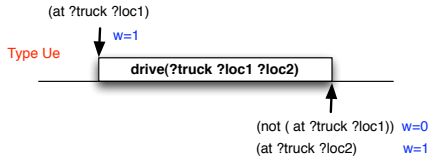


An unbalanced operator of type Us is safe if it is mutex with all those operators that may increase the weight of γ over its duration. The following picture shows a simple example of when this might happen.

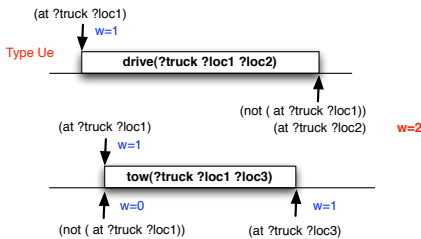


Unbalanced operators of type Us are particularly common because they model the usage of renewable resources. A renewable resource is needed during the execution of the action, so the weight goes from one to zero at start, but it is not consumed by the action, so the weight returns to one at end.

- **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 $\mathcal{C} = \{\{at(truck, loc)\}, \{truck\}, \{loc\}\}$.



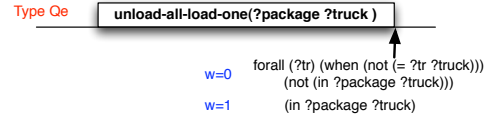
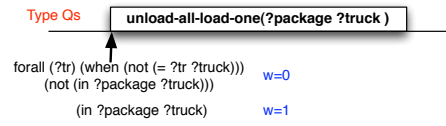
An unbalanced operator of type Ue is safe if it is mutex with all operators that may alter the weight during its execution. Although this operator does not cause an overall change in the weight of γ when executed in isolation, it might give rise to problematic situations when another operator op_i capable 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.



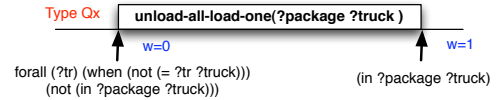
One could argue that unbalanced operators of type Ue originate from a faulty description of renewable resources and so they should in reality be operators of type Us. We have found a few examples of operators of type Ue in the domains of previous IPCs, but none resulted in provable invariants.

6. **Type Q:** **Quantified Delete.** The operator op sets the weight of γ 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:

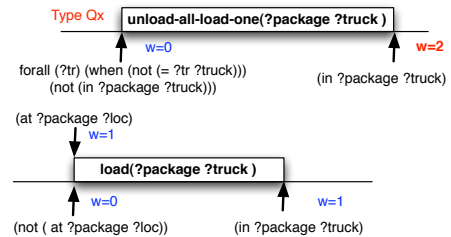
- **Type Qs and Qe:** a universally quantified effect sets the weight to zero at some time-point (start/end) and an add effect increases the weight by one at the same time-point (start/end). Operators of type Qs and Qe are safe because they ensure that only the single add effect will be true. The figure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



- **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 $\mathcal{C} = \{\{in(package, truck)\}, \{package\}, \{truck\}\}$.



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 $S = \{0, 1\}$ instead of $S = \{1\}$, 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:
 $\exists p \in (\text{Cond}_{start}(op_1) \cup \text{Cond}_{all}(op_1) \cup \text{Eff}_{start}(op_1)) :$
 $\neg p \in (\text{Cond}_{start}(op_2) \cup \text{Cond}_{all}(op_2) \cup \text{Eff}_{start}(op_2))$
2. **End-End:** op_1 and op_2 cannot end at the same time if:
 $\exists p \in (\text{Cond}_{end}(op_1) \cup \text{Cond}_{all}(op_1) \cup \text{Eff}_{end}(op_1)) :$
 $\neg p \in (\text{Cond}_{end}(op_2) \cup \text{Cond}_{all}(op_2) \cup \text{Eff}_{end}(op_2))$
3. **Start-End:** op_1 cannot start at the time that op_2 ends if:
 $\exists p \in (\text{Cond}_{start}(op_1) \cup \text{Cond}_{all}(op_1) \cup \text{Eff}_{start}(op_1)) :$
 $\neg p \in (\text{Cond}_{end}(op_2) \cup \text{Eff}_{end}(op_2))$
4. **Invariant-Start:** op_2 cannot start during op_1 if:
 $\exists p \in \text{Cond}_{all}(op_1) :$
 $\neg p \in (\text{Cond}_{start}(op_2) \cup \text{Cond}_{all}(op_2) \cup \text{Eff}_{start}(op_2))$
5. **Invariant-End:** op_2 cannot end during op_1 if:
 $\exists p \in \text{Cond}_{all}(op_1) :$
 $\neg p \in (\text{Cond}_{end}(op_2) \cup \text{Cond}_{all}(op_2) \cup \text{Eff}_{end}(op_2))$
6. **Invariant-Invariant:** op_1 and op_2 cannot overlap if:
 $\exists p \in \text{Cond}_{all}(op_1) : \neg p \in \text{Cond}_{all}(op_2)$

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 increasing operator of type Ix is safe if it is:
 1. mutex IS with any operator of type $(I,Q)s$
 2. mutex IE with any operator of type $Us, (I,Q)x,$ and $(I,Q)e$
- An unbalanced operator of type Us is safe if it is:
 1. mutex IS with any operator of type $(I,Q)s$
 2. mutex IE with any operator of type $Us, (I,Q)x, (I,Q)e$
- An unbalanced operator of type Ue is safe if it is:
 1. mutex IS with any operator of type $(I,Q,B)s$
 2. mutex IE with any operator of type $Us, (I,Q)x,$ and $(I,Q,B,U)e$
- A quantified delete operator of type Qx is safe if it is:
 1. mutex IS with any operator of type $(I,Q)s$
 2. mutex IE with any operator of type $Us, (I,Q)x,$ and $(I,Q,U)e$

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 γ 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 five 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 (Gerevini and Schubert, 2000; 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

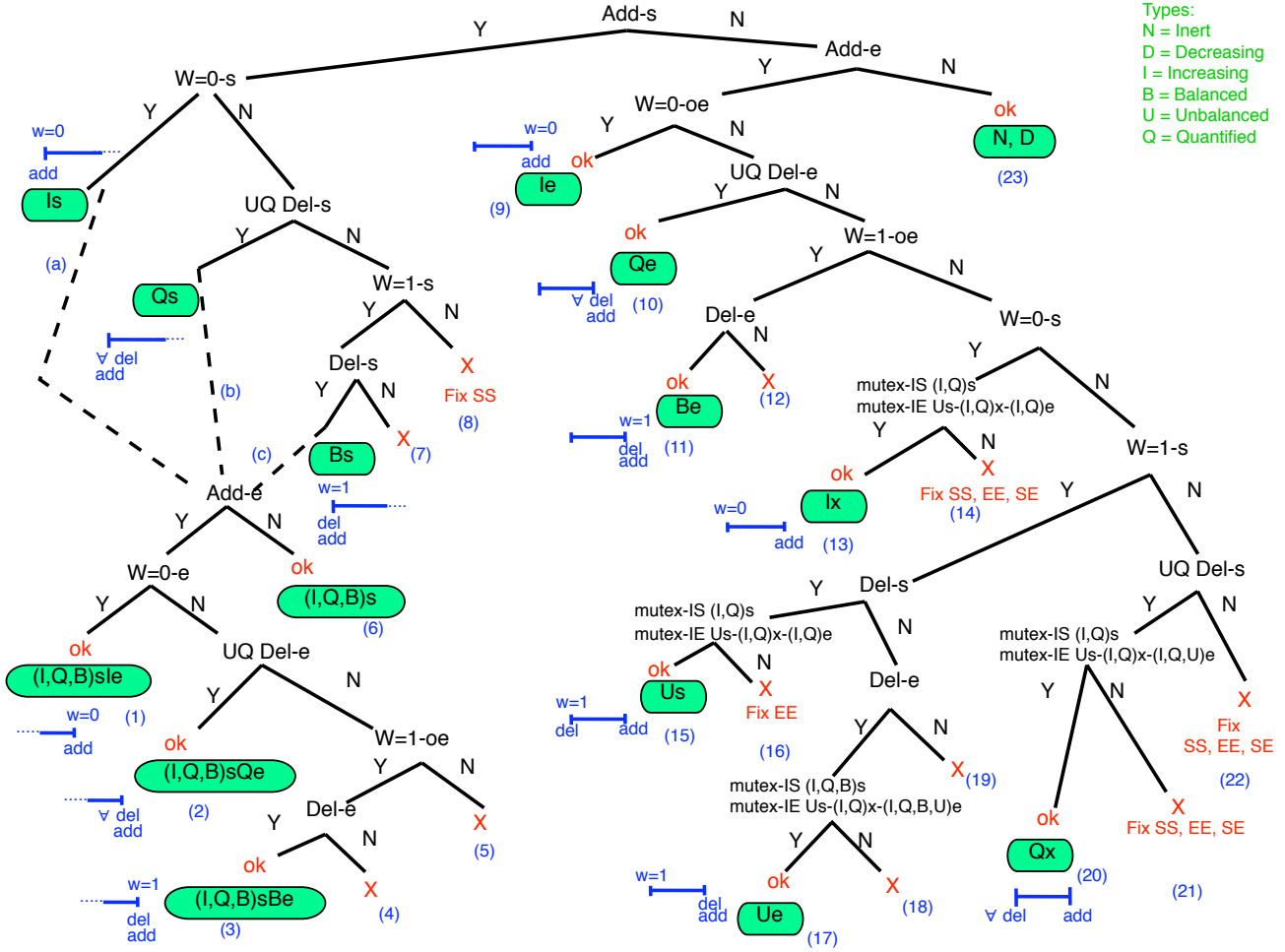


Figure 1: Decision Tree \mathcal{T} for checking whether an operator op is safe w.r.t an instance γ of a candidate \mathcal{C} .

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 (Edelkamp and Helmert, 1999), 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 Φ contains only one atom ϕ and the set V contains only one counted variable. The candidate $\mathcal{C}_{at} = \{\{at(truck, loc)\}, \{truck\}, \{loc\}\}$ 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} .
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

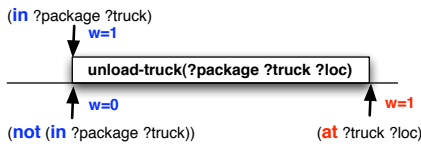
The choice of how to fix a failed candidate depends on the features of the operators that threaten it. More specifically, given a candidate $\mathcal{C} = \langle \Phi, F, V \rangle$ that has been rejected because it is threatened by an operator op , we refine \mathcal{C} by picking a new atom ϕ , which is chosen on the basis of the structure of op as explained below, and adding it to the components' set Φ of \mathcal{C} . So, we obtain the new candidate $\mathcal{C}' = \langle \{\Phi \cup \phi\}, F', V' \rangle$, which will not fail for the same reasons as \mathcal{C} , but might fail for different reasons. The new atom ϕ 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 ϕ unifies with a positive condition at start and a delete effect at start of op .
2. **Fix EE:** the atom ϕ unifies with a positive condition at end (or over all) and a delete effect at end of op .
3. **Fix SE:** the atom ϕ unifies with a positive condition at start and a delete effect at end of op .

Given a candidate \mathcal{C} and an instance γ , 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 γ , but no delete effects or conditions involving γ 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 $S = \{0, 1\}$ 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 $S = \{0, 1\}$ 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 $\mathcal{C}_{at} = \langle \{at(package, loc)\}, \{package\}, \{loc\} \rangle$, 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 actually 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.

If a candidate \mathcal{C} is discarded because an operator op of type Us or Qx does not satisfy the mutex checks as requested, we have two additional options for fixing \mathcal{C} . In particular, if op is of type Us and is not mutex-IS with an operator op_i of type Is or Qs, then we create new candidates by picking an atom ϕ that must involve only the variables in F and at most one other variable and satisfy the following criteria:

4. **Fix M-SS:** ϕ unifies with a positive condition at start and a delete effect at start of op_i . In such a way, op_i becomes of type Bs.

If op is not mutex-IE with an operator op_i of type (I,Q)e or (I,Q)x, then we create new candidates by: i) applying Fix M-SS so as to make op_i of type Us and ii) picking an atom ϕ that must involve only the variables in F and at most one other variable and satisfy the following criterium:

5. **Fix M-EE:** the atom ϕ unifies with a positive condition at end (or over all) and a delete effect at end of op_i . In such a way op_i becomes of type Be.

Finally, if op is of type Qx and is not mutex-IS with an operator op_i of type Is or Qs, then we create new candidates by applying Fix M-SS; if op is of type Qx and is not mutex-IE with an operator op_i of type (I,Q)e or (I,Q)x, then we create new candidates by applying Fix M-EE.

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.

Figure 2 presents the invariants that the algorithm finds for the temporal domains of the IPC-6 (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 positions 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, $\{in\ 0\ [1],\ at\ 0\ [1]\}$ indicates the invariant having $\{at(package, location)\}$ in $\{package, vehicle\}$ as components, $package$ as a fixed variable, and $\{location, vehicle\}$ as counted variables. Next

to each invariant, we report how many operators of each type we found during the synthesis of that invariant. The most common cases are: 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 a few operators of type 6c, which correspond to balanced operators at start. If an invariant was obtained by applying a fix, then we report what type of fix was used. From Figure 2, we can conclude that the temporal domains of the IPC-6 are fairly well constructed. There are no instances of operators of type Ue, which would likely correspond to malformed renewable resources, and there are not many domains that include univariably quantified conditions or effects.

In examining additional domains from previous IPCs, we have seen more variability in the types of operators. We found operators of types 6c, 11, 15, and 23. Additional operators of types 8, 12, 16, 17, 18, and 22 were found while examining candidate invariants that were ultimately rejected. Based on discussions with Will Cushing², these findings appear to be consistent with his analysis (Cushing et al., 2007).

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. In particular, it uses the cardinality set $S = \{1\}$ instead of cardinality set $S = \{0, 1\}$ and does not perform any mutex checks, which means that it considers safe only balanced operators of type Bs and Be.

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.

Finally, 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) just described, and our Temporal Invariant Synthesis (TIS). In many domains the TIS produces 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 $S = \{0, 1\}$ instead of $S = \{1\}$ and by analyzing

²Personal communication

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 IPC-6, IPC-5, IPC-4 and IPC-3 by using the Temporal Invariant Synthesis (TIS) and the Simple Invariant Synthesis (SIS).

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, 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

woodworking-numeric: {unused 0, wood 0 [1]} t-15: 2, t-23: 14 Fix SS {unused 0, treatment 0 [1]} t-15: 7, t-23: 9 Fix SS {unused 0, surface-condition 0 [1]} t-15: 4, t-23: 12 Fix SS {unused [0]} t-23: 16 {unused 0} t-23: 16 {idle [0]} t-15: 9, t-23: 7 {idle 0} t-15: 9, t-23: 7	modeltrain-numeric: {next-train 1 [0], first-train-in-head-segment 0} t-6c: 2, t-23: 384 Fix SS {switch-exit 0 [1]} t-15: 1, t-23: 385 {switch-entrance 0 [1]} t-15: 1, t-23: 385 {tail-segment 0 [1]} t-6c: 2, t-23: 384 {head-segment 0 [1]} t-6c: 2, t-23: 384 {idle [0]} t-15: 8, t-23: 378 {idle 0} t-15: 8, t-23: 378	openstacks-adl: {stacks-avail [0]} t-15: 2, t-23: 43 {waiting [0], started [0], shipped [0]} t-15: 2, t-23: 43 Fix SS {waiting 0, started 0, shipped 0} t-15: 2, t-23: 43 Fix SS {started [0], waiting [0]} t-15: 4, t-23: 44 Fix SS {started 0, waiting 0} t-15: 1, t-23: 44 Fix SS {waiting [0]} t-23: 45 {waiting 0} t-23: 45
parcprinter-strips: {uninitialized } t-23: 41 {timepoint 0 [1], location 0 [1]} t-15: 26, t-23: 15 Fix SS {notprintedwith 0 1 [2]} t-23: 41 {notprintedwith 0 2 [1]} t-23: 41 {notprintedwith 1 2 [0]} t-23: 41 {notprintedwith 0 1 2} t-23: 41 {hasimage 0 1 [2], notprintedwith 0 1 [2]} t-15: 4, t-23: 37 Fix SS	sokoban-strips: {at 0 [1]} t-15: 3, t-23: 269 {at 1 [0], clear 0} t-15: 3, t-23: 269 Fix SS {clear [0]} t-15: 3, t-23: 269	crewplanning-strips: {unused [0]} t-15: 1, t-23: 22 {unused 0} t-15: 1, t-23: 22 {currentday 0 [1]} t-15: 1, t-23: 22
elevators-strips: {passengers 0 [1]} t-15: 2, t-23: 478 {lift-at 0 [1]} t-15: 4, t-23: 476 {passenger-at 0 [1], boarded 0 [1]} t-15: 2, t-23: 478 Fix SS	pegsol-strips: {free 0, occupied 0} t-15: 1, t-23: 31 Fix SS {occupied [0]} t-15: 1, t-23: 31	transport-numeric: {ready-loading [0]} t-15: 2, t-23: 64 {ready-loading 0} t-15: 2, t-23: 64 {in 0 [1], at 0 [1]} t-15: 3, t-23: 63 Fix SS

Figure 2: Invariants for the temporal domains of the IPC-6.

Domains	# SV		
	NIS	SIS	TIS
Crew Planning - p10	112	112	106
Crew Planning - p20	305	305	261
Crew Planning - p30	510	510	498
Elevators-Num - p10	193	193	21
Elevators-Num - p20	578	578	34
Elevators-Num - p30	1216	1216	49
Elevators-Str - p10	203	203	21
Elevators-Str - p20	592	592	34
Elevators-Str - p30	1240	1240	49
Openstacks-Adl - p10	97	97	57
Openstacks-Adl - p20	166	166	97
Openstacks-Adl - p30	235	235	137
Openstacks-Num - p10	71	71	29
Openstacks-Num - p20	121	121	49
Openstacks-Num - p30	171	171	69
Openstacks-Num-Adl - p10	85	85	57
Openstacks-Num-Adl - p20	145	145	97
Openstacks-Num-Adl - p30	205	205	137
Openstacks-Str - p10	83	83	29
Openstacks-Str - p20	142	142	49
Openstacks-Str - p30	201	201	69

Domains	# SV		
	NIS	SIS	TIS
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).

$\mathcal{C} = \langle \{\text{pred}(\text{arg1}, \text{arg2})\}, \{\text{arg1} - \text{supertype}\}, \{\text{arg2} - \text{type}\} \rangle$, 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 $\mathcal{C}_1 = \langle \{\text{pred}(\text{arg1}, \text{arg2})\}, \{\text{arg1} - \text{subtype1}\}, \{\text{arg2} - \text{type}\} \rangle$ and $\mathcal{C}_2 = \langle \{\text{pred}(\text{arg1}, \text{arg2})\}, \{\text{arg1} - \text{subtype2}\}, \{\text{arg2} - \text{type}\} \rangle$. 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 (ICAPS'08)*.
- Blum, A., and Furst, M. 1997. Fast planning through planning graph analysis. *Artificial Intelligence* 90:281–300.
- 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 (ICAPS-07)*, 105–112.
- Edelkamp, S., and Helmert, M. 1999. Exhibiting knowledge in planning problems to minimize state encoding length. In *Proc. of the Fifth European Conference on Planning (ECP'99)*, 135–147.
- Edelkamp, S., and Hoffmann, J. 2004. PDDL2.2: The language for the classical part of the 4th international planning competition. Technical Report 195, Albert-Ludwigs-Universität Freiburg.
- Fox, M., and Long, D. 1998. The automatic inference of state invariants in tim. *Journal of Artificial Intelligence Research* 9:367421.
- Fox, M., and Long, D. 2003. PDDL 2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research* 20:61–124.
- 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.
- Gerevini, A., and Schubert, L. 2000. Discovering state constraints in discoplan: Some new results. In *In Proc. of the 17th National Conference on Artificial Intelligence (AAAI-2000)*, 761–767.
- 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 (AIPS-94)*, 61–67. AAAI Press.
- 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.
- Kautz, H., and Selman, B. 1999. Unifying sat-based and graph-based planning. In *Proc. of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99)*, 318–325. Morgan Kaufmann Publishers Inc.
- Muscettola, N. 1994. HSTS: Integrating planning and scheduling. In Zweben, M., and Fox, M., eds., *Intelligent Scheduling*. Morgan Kauffmann. 451–469.
- Rintanen, J. 2000. An iterative algorithm for synthesizing invariants. In *Proc. of the Seventeenth National Conference on Artificial Intel ligen ce (AAAI-2000)*, 806–811.
- Smith, D., and Jónsson, A. 2002. The logic of reachability. In *Proc. of the Sixth International Conference on AI Planning and Scheduling (AIPS-02)*, 379–387.