Handling of Inertia in a Planning System

Jana Koehler Jörg Hoffmann
Institute for Computer Science
Albert Ludwigs University
Am Flughafen 17
79110 Freiburg, Germany
<last-name>@informatik.uni-freiburg.de

TECHNICAL REPORT No. 122

Abstract

The generation of the set of all ground actions for a given set of ADL operators, which are allowed to have conditional effects and preconditions that can be represented using arbitrary first-order formulas is a complex process which heavily influences the performance of any planner or pre-planning analysis method.

The paper describes a sophisticated instantiation procedure that determines so-called *inertia* in a given problem representation and uses them to perform simplifications of formulas during the instantiation process. As a result, many inapplicable actions are detected and ruled out from the domain representation yielding a much smaller search space for the planner.

May 1999

1 Introduction

A planning system that handles a more expressive language than STRIPS requires sophisticated algorithmic solutions to quite a number of problems, which have nothing to do with the actual search process for a plan. One of these problems concerns the computation of the set of *actions* as all ground instances of a given set of operators.

The aim of the instantiation process is to generate all those ground instances of the planning operators that are applicable in some legal world state. This means, that the precondition of the operator should be satisfiable and its effects should be consistent. On one hand, a naive instantiation procedure that simply expands quantifiers and enumerates all possible instantiations of operator parameters will quickly render even simple planning problems unsolvable. On the other hand, a rather sophisticated instantiation procedure can rule out many actions, which will never be applicable in any reachable world state or that would—if applied—yield an inconsistent state. It should also return the most simple syntactic representation of preconditions and effects.

Many planning systems do generate the complete set of actions before planning actually starts. They use this set either for the encoding of the domain in other representation formalisms such as SAT [Kautz and Selman, 1996] or for the derivation of useful information that can help during planning, e.g., distance heuristics [Geffner, 1999], symmetries [Fox and Long, 1999], relevant actions [Nebel et al., 1997], and goal orderings [Koehler, 1998].

When using the PDDL language [McDermott et al., 1998] to represent ADL operators [Pednault, 1989], quite complex descriptions of preconditions and effects are possible:

- arbitrary function-symbol free first-order logic formulas represent preconditions,
- conditional effects have the form *(when antecedent consequent)* where the antecedent can be an arbitrary precondition and the consequent is a conjunction of literals, i.e., an atom either occurs positively or negatively in it. A conditional effect can also be universally quantified.

Given such an operator, the instantiation has to replace all occurring variables, which are either quantified or occur as parameters of the operator, by those type-consistent constants, which have been declared in the planning problem. In order to replace all variables by constants, the instantiation process proceeds in three phases:

1. the expansion of universal and existential quantifiers occurring in the first-order formulas eliminates most of the quantified variables,

2. the expansion of universally quantified conditional effects eliminates the remaining quantified variables,
3. the instantiation of operator parameters eliminates the variable parameters.[1]

In each phase, the following *atomic instantiation task* occurs:

> *Given a variable $?x$, a constant c and an atomic formula p, determine the resulting instantiation $p[?x/c]$.*

This is a trivial problem per se. But after having determined $p[?x/c]$ one can sometimes simplify this atomic formula to FALSE or TRUE, which in turn often leads to a further simplification of the operator representation. This is the topic of this paper, where we describe what kind of *atomic simplifications* are performed in IPP [Koehler et al., 1997] under which conditions, how this process can be efficiently implemented and how it affects the search space of the planning system.

The paper is organized as follows: Section 2 gives an overview of the three phases of the instantiation process. Section 3 defines the notion of *inertia* predicates and describes how the knowledge about inertia is used to perform *atomic simplifications*. It proves their soundness and describes how the underlying tests can be efficiently implemented. Section 4 defines how atomic simplifications can be propagated over the operator description to further simplify their representation. Section 5 shows how unary inertia relations can be encoded as *types* to speed up the instantiation process. Finally, in Section 6, the impact of the instantiation process on the search space of IPP is exemplified.

2 Overview over the Instantiation Process

After having parsed the domain and problem file into some appropriate data structure, a basic preprocessing step renames all variables in the logical formulas and assigns unique names to them. For example, the formula $\psi(?x) \wedge \forall\, ?x\, \varphi(?x)$ is equivalently transformed into $\psi(?x_1) \wedge \forall\, ?x_2\, \varphi(?x_2)$. Then code tables are generated, which map strings to unique numbers, i.e., we obtain one number for each predicate name, variable name, and constant name. Internally, all subsequently described operations work over trees of numbers representing the formulas.

Figure 1 shows the precondition of the **remove** operator from the *assembly* domain [McDermott, 1998] with the quantifiers in frames and the underlined *requires* predicate, which will be used throughout this paper to illustrate the instantiation process. This predicate has two arguments, the first one *?whole*

[1] In IPP the assumption is made that different operator parameters are instantiated with different constants, i.e., the planner never generates actions like *move(a,a)*.

being an operator parameter of type *assembly* and the second one *?res* being a universally quantified variable of type *resource*.

```
:action remove
:parameters (?part ?whole - assembly)
:vars (?res - resource)
:precondition
(and ( forall (?res - resource)
              (imply (requires ?whole ?res) (committed ?res ?whole)))
     (incorporated ?part ?whole)
     (or      (and (transient-part ?part ?whole)
                   ( forall (?prev - assembly)
                            (imply (remove-order ?prev ?part ?whole)
                                   (incorporated ?prev ?whole))))
              (and (part-of ?part ?whole)
                   (not ( exists (?prev - assembly)
                           (and (assemble-order ?prev ?part ?whole)
                                (incorporated ?prev ?whole)))))))
```

Figure 1: Precondition of the **remove** operator from the *assembly* domain.

The schematic tree-like representation of this first-order formula is shown in Figure 2. The leaves of the tree contain the atomic formulas. IPP's instantiation process traverses the trees top-down and expands quantifiers one after the other, i.e., it reaches the first quantifier *forall (?res - resource)* and extracts the variable $?res$ together with its type *resource*. From the problem file, IPP knows all constants of this type. These are now used to instantiate $?res$.

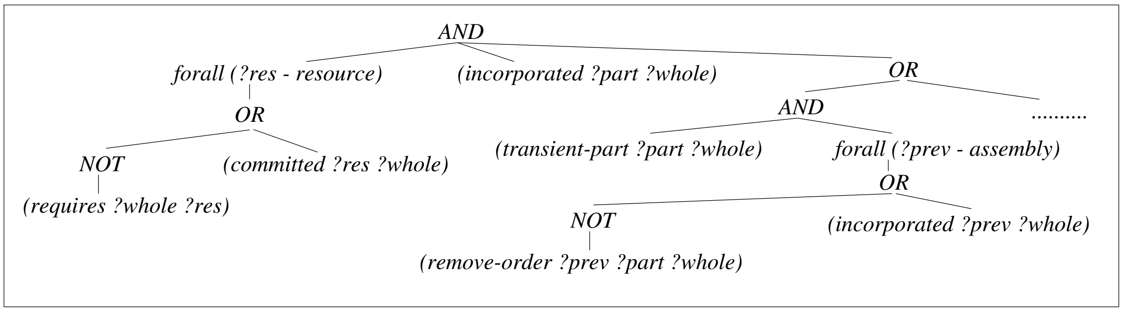

Figure 2: Tree representation of ADL formulas. Note that formulas of the form $\varphi \rightarrow \psi$ have been replaced with the equivalent $\neg\varphi \vee \psi$ already during the parsing process.

The process considers all constants one after the other. For each constant, a copy of the subtree representing the quantified formula is generated. In the leaves of this tree, all occurrences of $?res$ are replaced by the selected constant. As we will see in Section 3, this can lead to so-called *atomic simplifications*, which replace an atomic formula by either TRUE or FALSE. In turn, the whole tree can sometimes be simplified to TRUE or FALSE, see Section 4 for a detailed description.

In the case of a universal quantifier, the resulting trees are joined by an AND. In the case of an existential quantifier, the trees are joined by an OR. Figure 3 illustrates the result of the process.

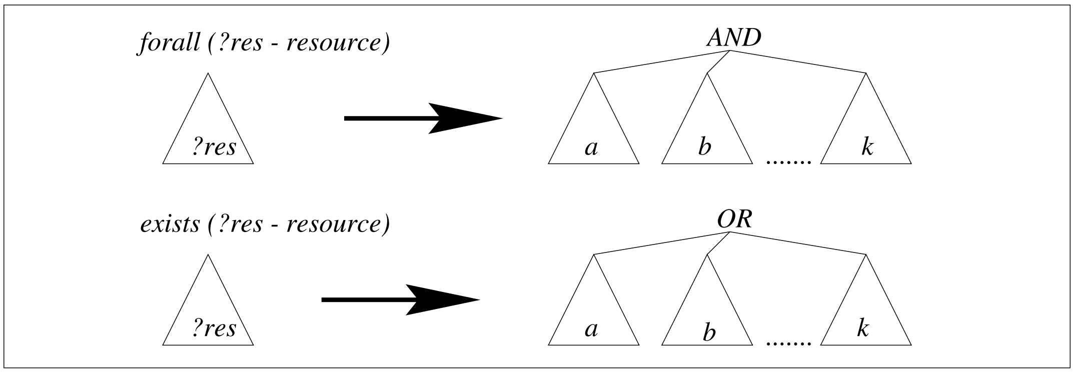

Figure 3: Copies of trees generated during the expansion of quantifiers. Obviously, if one of the subtrees resulting from the expansion of a universal (existential) quantifier can be simplified to FALSE (TRUE), then the whole formula can be simplified to FALSE (TRUE).

The expansion of quantified conditional effects proceeds in a similar way. Figure 4 shows the tree representation of the **move** operator from the *briefcase* domain, whose conditional effect contains the quantifier prefix *forall (?x - object)*. The copied trees will now also contain *when* nodes, i.e., numerous partially instantiated copies of the conditional effect are generated.

The process for instantiating the parameters of an operator fits into the same scheme. In each step, it takes a variable parameter together with the set of type-consistent constants. For each of these constants, a copy of the tree representing the operator is generated, and each occurrence of the parameter in this tree is replaced with the constant. Then, the operator tree is simplified. If, for example, it's precondition simplifies to FALSE, the whole partially instantiated operator can be skipped and removed from the domain. After all parameters have been instantiated, each tree represents a ground instance (an action) of the operator.

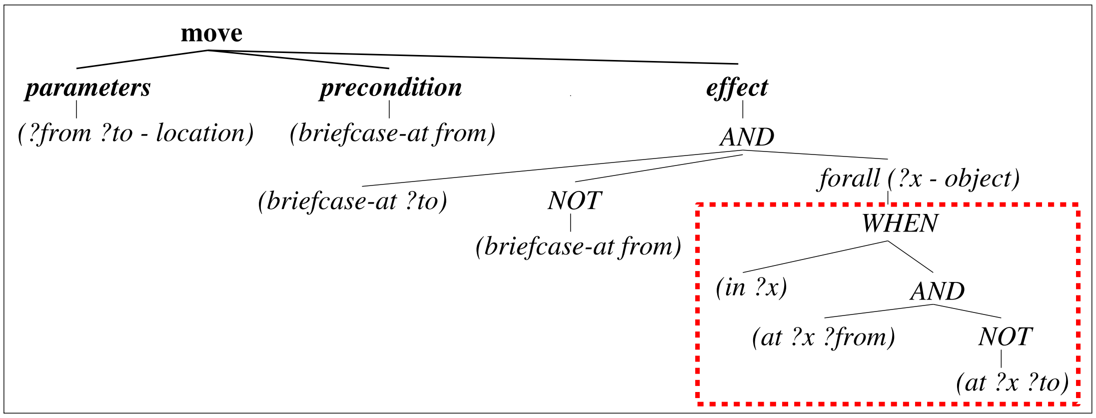

Figure 4: Tree-Representation of an operator with a quantified conditional effect. Expanding the quantifier *forall (?x - object)* results in copies of the tree starting in the *when* node.

3 Identification of Inertia and their Use during the Instantiation

The tree-copying process takes a variable $?x$ and a constant c as input and traverses the subformula represented in the tree. Whenever it reaches an atomic formula p, it gets replaced with $p[?x/c]$. In many situations, it is worthwhile to invest some more effort at this point and have a closer look at the result of the instantiation. Under certain conditions, namely if p represents an *inertia* relation, one can determine that $p[?x/c]$ must either always be TRUE or FALSE. This can even be the case if $p[?x/c]$ is not yet fully instantiated. Let us consider an example from the *assembly* domain. The object declaration introduces a list of objects followed by their types:

:objects *doodad valve frob sprocket socket plug - assembly*
charger voltmeter battery - resource

The specification of the initial state contains the following instances of the *requires* relation:

:init *(requires frob charger) (requires sprocket charger) (requires socket voltmeter)*
(requires doodad voltmeter) (requires plug voltmeter)

Given the number of declared constants for the two types, the *requires* relation can be instantiated with $6 \times 3 = 18$ different type-consistent tuples, of which 5 occur in the initial state.

The expansion of the first universal quantifier that is shown framed in Figure 1 generates three copies of the formula tree, each containing either the partially instantiated atom *(requires ?whole voltmeter)*, *(requires ?whole charger)*, or *(requires ?whole battery)*. Two observations can be made:

- If *(requires ?whole ?res) never occurs as a positive effect* of any operator then the only instances of this predicate, which can hold in any state, are those that are specified in the initial state. This, for example, implies that *(requires ?whole battery)* can never hold and is therefore equivalent to FALSE.

- If *(requires ?whole ?res) never occurs as a negative effect* of any operator then the only instances that can be FALSE in any state are those that are *not* contained in the initial state. Now, if the initial state contained all possible ground instances of, say, *(requires ?whole voltmeter)*, then this partially instantiated predicate could be replaced by TRUE. All of its instances would be initially true and thus persist in all reachable states.

In the following, we will formalize these ideas and give a precise notion of *inertia*.

3.1 Inertia Relations

IPP proceeds over the domain and problem description and collects all used relation names. For each relation it checks if it satisfies one of the following definitions:

Definition 1 *A relation is a* positive inertia *iff it does not occur positively in an unconditional effect or the consequent of a conditional effect of an operator.*

Definition 2 *A relation is a* negative inertia *iff it does not occur negatively in an unconditional effect or the consequent of a conditional effect of an operator.*

Relations, which are positive as well as negative inertia, are simply called *inertia*. Relations, which are neither positive nor negative inertia, are called *fluents*. The detection of inertia and fluents is easy because in ADL, effects are restricted to conjunctions of literals. Furthermore, this information can be obtained with a single pass over the domain description, which takes almost no time at all. In the assembly domain, the status of all relations can be inferred as shown in Figure 5.

number	predicate name	positive effect	negative effect	status
0	available	yes	yes	fluent
1	requires	no	no	inertia
2	part-of	no	no	inertia
3	transient-part	no	no	inertia
4	assemble-order	no	no	inertia
5	remove-order	no	no	inertia
6	complete	yes	no	negative inertia
7	committed	yes	yes	fluent
8	incorporated	yes	yes	fluent

Figure 5: Inertia Relations in the Assembly Domain.

3.2 Atomic Simplifications

In order to decide if an inertia can be replaced by TRUE or FALSE one needs to determine and count all type-consistent ground instances of an inertia predicate p that match a partially instantiated occurrence of p.

Definition 3 *Let* τ *be some* type name.

$$dom(\tau) = \{c_1, \ldots, c_m\}$$

denotes the domain *of* τ, *i.e., the set of constants having type* τ.

In PDDL, each constant is either explicitly declared as being of a particular type or it has the default type *object*. The same applies to all operator parameters or quantified variables. Each predicate must be explicitly declared together with its arguments, for which type names can be given or the default type is assumed.

Definition 4 *Let* p *be a predicate of arity* n. *Let* $\vec{a} = (a_1, \ldots, a_n)$ *be the argument vector of some partially instantiated occurrence of* p *where each* a_i *is either a constant or variable. With*

$$V(\vec{a}) = \{i \in \{1, \ldots, n\} \mid a_i \textit{ is a variable}\}$$

we denote the positions in $\vec{a}$ *that are occupied by variables.*

Definition 5 *Let* p *be a predicate and let* $\vec{a}$ *be the argument vector of some partially instantiated occurrence of* p. *Let* τ_i *be the type name of position* i *in predicate* p. *Then*

$$\mathsf{MAX}(p\ \vec{a}) = \prod_{i \in V(\vec{a})} |dom(\tau_i)|$$

denotes the number of all *possible type-consistent ground instances of p that unify with the argument vector* $\vec{a}$*. In contrast,*

$$\mathsf{N}(p\ \vec{a}) = |\{(p\ \vec{c}) \in \mathcal{I} \mid (p\ \vec{c}) \textit{ unifies with } (p\ \vec{a})\}|$$

denotes the number of unifying ground instances of p that are contained in the initial state $\mathcal{I}$*. Obviously,* $\mathsf{N}(p\ \vec{a}) \leq \mathsf{MAX}(p\ \vec{a})$ *holds.*

It is worthwhile noticing here that IPP will remove all variables or parameters that have an *empty type*, see Section 4 for more details. Therefore, we have $|dom(\tau_i)| \neq 0$ for each position i of any partially instantiated occurrence of the predicate p. Thus, for any $(p\ \vec{a})$ holds $\mathsf{MAX}(p\ \vec{a}) \neq 0$. As an example, let us consider $(p\ \vec{a})$ =*(requires ?whole voltmeter)*, for which one obtains

$V(\vec{a})$	=	$\{1\}$ /* only one variable argument */
τ_1	=	*assembly*
dom(*assembly*)	=	{doodad, valve, frob, sprocket, socket, plug}
MAX(*requires ?whole voltmeter*)	=	6 /* 6 objects can instantiate *?whole* */
N(*requires ?whole voltmeter*)	=	3 /* 3 instances in the initial state contain *voltmeter* */

A partially instantiated atomic formula can be simplified to TRUE or FALSE if it satisfies one of the conditions defined below.

Definition 6 *Let* $(p\ \vec{a})$ *be some partially instantiated atomic formula constructed during the instantiation process.*

If *p is a positive inertia and* $\mathsf{N}(p\ \vec{a}) = 0$
then $(p\ \vec{a})$ *is simplified to* FALSE*.*

If *p is a negative inertia and* $\mathsf{N}(p\ \vec{a}) = \mathsf{MAX}(p\ \vec{a})$
then $(p\ \vec{a})$ *is simplified to* TRUE*.*

In all other cases $(p\ \vec{a})$ *cannot (yet) be simplified and remains in the formula tree as it is.*

From the treatment of empty types, we know that $\mathsf{MAX}(p\ \vec{a}) \neq 0$ holds for $(p\ \vec{a})$. Therefore, obviously at most one of the above tests can succeed. For example, *(requires ?whole battery)* is a positive inertia. It can be simplified to FALSE because no *requires* instance from the initial state matches the argument vector (*?whole, battery*), i.e., N*(requires ?whole battery)*$= 0$ and the first test succeeds.

That an atomic formula can sometimes be simplified to TRUE is best seen in the case when it is fully instantiated. Take, for example, *(requires plug voltmeter)*. This fact occurs in the initial state, so N*(requires plug voltmeter)*$= 1 \neq 0$ and the first test fails. However, MAX*(requires plug voltmeter)*$= \prod_{i \in \emptyset} |dom(\tau_i)| = 1$ and the second test succeeds. This reflects that *(requires plug voltmeter)* is initially TRUE and will never be made FALSE because *requires* is a negative inertia.

Theorem 1 (Soundness of Simplifications)
Given a planning domain and problem, if $(p\ \vec{a})$ *is simplified to*
(1) FALSE*, then no state s which is reachable from the initial state satisfies any type-consistent ground instance of* $(p\ \vec{a})$*.*
(2) TRUE*, then any state s which is reachable from the initial state satisfies all type-consistent ground instances of* $(p\ \vec{a})$*.*

Proof:
(1) holds because if $\mathsf{N}(p\ \vec{a}) = 0$ then none of the type-consistent ground instances of $(p\ \vec{a})$ are contained the initial state. Since p is a positive inertia, no other instances can be generated by any plan.
(2) holds because if $\mathsf{N}(p\ \vec{a}) = \mathsf{MAX}(p\ \vec{a})$ then all type-consistent ground instances are contained in the initial state and will persist in all reachable states because p is a negative inertia. ∎

Atomic simplification requires to determine the number $\mathsf{N}(p\ \vec{a})$ of all those ground tuples in the initial state that unify with a given argument vector of arbitrary length, containing variables or constants at arbitrary positions. Using a naive solution, this means to perform a single pass over the initial state $\mathcal{I}$, testing for each fact if it unifies with $(p\ \vec{a})$. Obviously, the time complexity is $\Theta(|\mathcal{I}| * n_{max})$ where n_{max} denotes the maximum arity of the predicates. The test for atomic simplification has to be done for every leaf of every tree that is ever generated during the instantiation process. The number of these leaves is likely to be enormous, so there is a strong need for a highly efficient method to find $\mathsf{N}(p\ \vec{a})$. In the following, such a method is described, which allows to retrieve the number of matching initial facts in time linear in the *length of* $\vec{a}$, i.e., in the arity of the predicates, $O(n_{max})$.

3.3 Efficient Implementation of Atomic Simplifications

In principle, the idea behind the implementation is as simple as this: Before instantiation starts, perform a single pass over the initial state and create tables in which the occurring tuples are documented. Then later determine the proper table entry for $(p\ \vec{a})$ and look up the correct value of $\mathsf{N}(p\ \vec{a})$. What makes the process complicated is that we have to deal with *partially instantiated* argument vectors $\vec{a}$.

Let us consider the *requires* predicate as an example. For its argument vector of length 2, four cases can occur:

(1) Both arguments are variables and thus $\vec{a} = (?x_1, ?x_2)$. One needs to determine the total number of occurrences of *requires* (with arbitrary arguments) in the initial state.

(2) The first argument is instantiated, but the second argument is a variable and thus $\vec{a} = (c_1, ?x_2)$. We need the number of occurrences of *requires* where the first argument is c_1.

(3) Only the second argument is instantiated and $\vec{a} = (?x_1, c_2)$. We need to count the occurrences with c_2 at the second position.

(4) Both arguments are instantiated and $\vec{a} = (c_1, c_2)$. The question is whether the initial state contains *(requires c_1 c_2)*.

For each of these four cases, a separate table is constructed. The table entries are computed from the initial state. The dimension of each table corresponds to the number of instantiated positions of the argument vector. In Case (1), the table is therefore 0-dimensional and simply consists of an integer counting the number of *requires* facts in the initial state. For Cases (2) and (3), a 1-dimensional table is needed, with one entry for each object that is type-consistent with the instantiated argument. For each of these objects, the corresponding entry counts the number of times that *requires* occurred in the initial state instantiated with that object. In Case (4), a 2-dimensional table is constructed. Its entries are indexed by all pairs of type-consistent objects that can instantiate the *requires* predicate. For each such pair, the entry is set to 1 iff *requires* occurred in the initial state instantiated with that pair. All tables are shown in Figure 6.

$\emptyset$

5

$\{1\}$

doodad	1
valve	0
frob	1
sprocket	1
socket	1
plug	1

$\{2\}$

charge	2
voltmeter	3
battery	0

$\{1, 2\}$

	charger	voltmeter	battery
doodad	0	1	0
valve	0	0	0
frob	1	0	0
sprocket	1	0	0
socket	0	1	0
plug	0	1	0

Figure 6: The tables to represent those facts from $\mathcal{I}$ that match a given argument vector for the *requires* predicate. Given the set of instantiated positions as $\emptyset$, $\{1\}$, $\{2\}$ or $\{1, 2\}$, the corresponding tables are shown from left to right.

Let p be a predicate of arity n. For each subset $C \subseteq \{1, \ldots, n\}$, a table $\mathsf{T}(p\ C)$ has to be constructed. The table is $|C|$-dimensional and lists one entry $\mathsf{T}(p\ C)(\vec{c})$ for each type-consistent tuple $\vec{c}$ of constants that can possibly instantiate p at exactly the positions in C. All entries are initially set to zero.

Note that although the number of tables is exponential in the arity of the predicates, planning domain representations rarely use predicates with more than 3 or 4 arguments.

Definition 7 *Let $\vec{a} = (a_1, \ldots, a_n)$ be an argument vector of size n, each a_i being either a variable or a constant. Let $C = \{i_1, \ldots, i_k\}$ be a set of possible positions, i.e., $C \subseteq \{1, \ldots, n\}$, where the $i_1, \ldots i_k$ are ordered increasingly. With*

$$\vec{a}|_C := (a_{i_1}, \ldots, a_{i_k})$$

we denote the restriction *of $\vec{a}$ to the positions in C.*

Intuitively, the restriction of a vector to some set $C \subseteq \{1, \ldots, n\}$ is obtained by simply skipping all those positions that are not in C, but preserving the ordering of the arguments.

Now for each ground atom $(p\ \vec{c})$ that occurs in the initial state, the following is done:

$$\begin{array}{l} \textbf{for } \text{all sets } C \subseteq \{1, \ldots, n\} \textbf{ do} \\ \quad \text{increment } \mathsf{T}(p\ C)(\vec{c}|_C) \\ \textbf{endfor} \end{array} \tag{1}$$

Performing process (1), we count for all instances of p in the initial state how often each combination of constants occurs for arbitrary sets C of positions.

Definition 8 *Let p be a predicate of arity n. Let $\vec{a} = (a_1, \ldots, a_n)$ be the argument vector of some partially instantiated occurrence of p. With*

$$C(\vec{a}) := \{i \in \{1, \ldots, n\} \mid a_i \textit{ is a constant}\}$$

we denote the positions where $\vec{a}$ is instantiated.

During instantiation, given a partially instantiated predicate $(p\ \vec{a})$, we determine the set $C(\vec{a})$ of positions where the argument vector is occupied by constants. The appropriate table $\mathsf{T}(p\ C(\vec{a}))$ is the one corresponding to that set. The entry in this table that we want to access is the one indexed by the constants in the current argument vector $\vec{a}$, i.e., by the restriction $\vec{a}|_{C(\vec{a})}$ of $\vec{a}$ to its constants.

Theorem 2 (Soundness of Tables)
Let the tables T be the result of performing process (1) for each fact in the initial state. Then we have for each partially instantiated predicate $(p\ \vec{a})$:

$$\mathsf{N}(p\ \vec{a}) = \mathsf{T}(p\ C(\vec{a}))(\vec{a}|_{C(\vec{a})})$$

Proof:
Per definition, $\mathsf{N}(p\ \vec{a}) = |\{(p\ \vec{c}) \in \mathcal{I} \mid (p\ \vec{c}) \text{ unifies with } (p\ \vec{a})\}|$. We will show for each fact $(p\ \vec{c}) \in \mathcal{I}$: When process (1) works on $(p\ \vec{c})$,

$$\mathsf{T}(p\ C(\vec{a}))(\vec{a}|_{C(\vec{a})}) \text{ gets incremented} \Leftrightarrow (p\ c) \text{ unifies with } (p\ \vec{a}) \quad (*)$$

As process (1) is performed for each fact in $\mathcal{I}$, the proposition follows directly from $(*)$, which remains to be shown.

$\Rightarrow$: We prove the contraposition. Let $(p' \ \vec{c}')$ be a fact in $\mathcal{I}$ that does *not* unify with $(p \ \vec{a})$. If $p' \neq p$, process (1) never even considers the table $\mathsf{T}(p \ C(\vec{a}))$. Otherwise, one entry in this table gets incremented when the process reaches $C = C(\vec{a})$. But, as $\vec{c}'$ does not unify with $\vec{a}$, there is at least one constant in $\vec{c}'|_{C(\vec{a})}$ that is different from the corresponding constant in $\vec{a}|_{C(\vec{a})}$. Therefore, the table entry in $\mathsf{T}(p \ C(\vec{a}))$ that gets incremented is different from the one for $\vec{a}|_{C(\vec{a})}$.
$\Leftarrow$: Let $(p \ \vec{c}) \in \mathcal{I}$ be a fact that unifies with $(p \ \vec{a})$. When process (1), working on $(p \ \vec{c})$, reaches $C = C(\vec{a})$, the entry $\mathsf{T}(p \ C(\vec{a}))(\vec{c}|_{C(\vec{a})})$ gets incremented. As $\vec{c}$ is a ground instance that unifies with $\vec{a}$, we have $\vec{c}|_{C(\vec{a})} = \vec{a}|_{C(\vec{a})}$, so this entry is exactly $\mathsf{T}(p \ C(\vec{a}))(\vec{a}|_{C(\vec{a})})$. ■

During the instantiation process it remains to find the corresponding table entry in order to determine the correct value of $\mathsf{N}(p \ \vec{a})$. Since constants are internally kept as *numbers* they can in principle be used as indices into a table. However, to directly index into the tables, one would need to define tables of *arbitrary dimension*. Instead, the implementation uses an implicit representation of the tables. The appropriate address is computed by performing a sweep over the argument vector, which takes time $O(n_{max})$. As arities are usually small, this running time is very close to constant anyway.

3.4 Ground Level Inertia

So far we have only considered the *predicates* which are never made true or false by a planning operator. These were used to constrain the instantiation process. Once the set of all actions has been determined, one can similarly define the *ground facts* that are never made true or false by one of the actions.

Definition 9 *A ground fact is a* positive ground inertia *iff it does not occur positively in an unconditional effect or the consequent of a conditional effect of an action.*

Definition 10 *A ground fact is a* negative ground inertia *iff it does not occur negatively in an unconditional effect or the consequent of a conditional effect of an action.*

An initial fact, which is a negative ground inertia, is never made FALSE and thus always satisfied in all reachable world states. It can be removed from the state description. All its occurrences in the preconditions of actions and in the antecedents of conditional effects can be simplified to TRUE.

A fact, which is a positive ground inertia and *not* contained in the initial state, is never satisfied in any reachable world state. All its occurrences in the preconditions of actions and in the antecedents of conditional effects can be simplified to FALSE.

The remaining facts are fluents that, roughly speaking, can possibly change their truth value during the planning process. They are therefore relevant to the representation of the planning problem.

Definition 11 *A ground fact is* relevant *iff*

1. *it is an initial fact and not a negative ground inertia, or if*
2. *it is not an initial fact and not a positive ground inertia.*

Using the table which corresponds to the fully instantiated case of the process described in Section 3.3, one can find all relevant facts by performing a single sweep over the initial state and the effects of all actions.

The simplified actions and the set of all relevant facts are then used by IPP to generate a bitvector representation for all states and actions, where each relevant fact corresponds to a position in a bitvector.

4 Simplification of Operator Representations

As we have already mentioned in Section 2, the instantiation process creates copies of trees representing formulas and operators. These trees can be simplified if one of their subtrees has been simplified to TRUE or FALSE, which can result from the atomic simplifications performed during the instantiation process. As soon as such an atomic simplification has replaced an atomic formula by TRUE or FALSE, the subsequently described non-atomic simplification operations are performed.[2]

$$
\begin{array}{rcl@{\qquad\qquad}rcl}
\neg\mathsf{TRUE} & \equiv & \mathsf{FALSE} & \neg\mathsf{FALSE} & \equiv & \mathsf{TRUE} \\
\mathsf{TRUE} \wedge \varphi & \equiv & \varphi & \varphi \wedge \varphi & \equiv & \varphi \\
\mathsf{FALSE} \wedge \varphi & \equiv & \mathsf{FALSE} & \varphi \vee \varphi & \equiv & \varphi \\
\mathsf{TRUE} \vee \varphi & \equiv & \mathsf{TRUE} & \varphi \wedge \neg\varphi & \equiv & \mathsf{FALSE} \\
\mathsf{FALSE} \vee \varphi & \equiv & \varphi & \varphi \vee \neg\varphi & \equiv & \mathsf{TRUE}
\end{array}
$$

Figure 7: Implemented Simplifications for First-Order Formulas.

[2]They are also performed once directly after the parsing of the domain and problem file.

The first-order formulas which represent the preconditions of operators and the antecedents of conditional effects are simplified based on the well-known tautologies as shown in Figure 7. Besides this, IPP implements the following simplifications:

1. If a quantified variable does not occur in the quantified formula, the quantifier is removed, i.e., $\forall\ ?x\ \varphi(?y)$ is simplified to $\varphi(?y)$.[3]

2. If a quantified variable $?x$ has an *unknown type*, which has not been declared in the :types field of the domain file or if it has an *empty type*, for which no constant has been declared in the problem file, then the quantified formula is replaced by TRUE in the case of a universal quantifier and by FALSE in the case of an existential quantifier.

3. An equality between two identical variable names, $?x =?x$, is simplified to TRUE. An equality between two identical constants, $c_1 = c_1$, is also simplified to TRUE. If the constants are different, i.e., $c_1 = c_2$, the equality is simplified to FALSE. In a fully instantiated formula, all occurrences of equalities have been replaced by TRUE or FALSE.

The simplification of first-order formulas can reduce a whole precondition, antecedent or consequent to TRUE or FALSE. In this case, the operator description can be simplified:

1. If the antecedent of a conditional effect becomes FALSE, the conditional effect is removed from the operator. In this case, the effect is never applicable because it requires FALSE to hold, i.e., the state must be inconsistent.

2. If the antecedent of a conditional effect becomes TRUE, the conditional effect becomes unconditional.

3. If the consequent of a conditional effect becomes TRUE, the conditional effect is removed because it does not lead to any state transition.

4. If the precondition or the unconditional effect of an operator becomes FALSE, the whole operator is removed from the domain.

5. If an operator has only TRUE as its unconditional effect and no conditional effects, then the whole operator is removed.[4]

[3]Unused quantified variables will usually not appear in the initial domain description. They can, however, appear as a result of atomic simplifications.

[4]Removing effects or whole operators can possibly turn fluents into inertia, i.e., one could repeat the whole analysis procedure again. However, such a phenomenon was not observed in any planning domain and therefore it seems not worth to invest the effort into such a fixpoint computation.

In the final set of actions, to which no simplifications can be applied anymore, all unconditional effects are merged into a single conjunction of literals and all conditional effects with identical antecedents are merged into a single conditional effect.
IPP also implements various syntax checks that help to develop proper domain representations:

1. An operator is removed (a warning is issued, but planning continues) if an operator parameter is declared using an unknown or empty type.

2. A parameter is removed from the operator description (a warning is issued, but the operator remains in the set) if it is declared, but nowhere used in the preconditions or effects.[5]

IPP aborts the instantiation process if it encounters one of the following situations:

1. A predicate symbol is overloaded. PDDL requires the declaration of predicates, their arity and the types of their arguments. When parsing the domain and problem files, IPP verifies that all occurrences of a predicate meet the declaration.

2. An equality statement occurs in an unconditional effect or in the consequent of a conditional effect.

3. An equality statement has less or more than two arguments.

4. A variable occurs that is neither declared as a parameter nor bound by a quantifier.

5. A constant occurs that has not been declared in the problem file.

5 Encoding Unary Inertia as Types

Many domains, in particular *all* STRIPS domains used in the planning competition contain *unary inertia.* These are predicates of arity one, which satisfy Definitions 1 and 2 and thus do not occur in any of the effects. In other words, the set of constants c that can ever (and will always) satisfy $(p\ c)$ is exactly the set of constants occurring as the arguments of the instances of p in the initial state.

Obviously, this set can be seen as the encoding of *type* information because the single variable argument of p can only be instantiated with one of these constants if we want to obtain a possibly satisfiable atomic formula. As a matter of fact, in the STRIPS domains from the planning competition, all unary inertia where

[5]Just like unused quantifiers, this can also happen as a result of simplifications.

intended to provide implicit type information, as there are no explicit types given in classical STRIPS, see Figure 8 for an example.

```
:action load-truck
:parameters (?obj ?truck ?loc)
:precondition (and (obj ?obj) (truck ?truck) (location ?loc)
                   (at ?truck ?loc) (at ?obj ?loc))
:effect (and (not (at ?obj ?loc)) (in ?obj ?truck))
```

Figure 8: The **load-truck** operator from the *logistics* domain. Note the untyped parameters and the underlined unary inertia predicates that implicitly encode the type information.

One can easily make this implicit type information explicit and remove all unary inertia from the domain description. The previously described instantiation process that identifies and simplifies inertia will also achieve the desired simplification of unary inertia, because they are simply a special case wrt. the length of the argument vector. However, doing it this way, the algorithm repeatedly generates copies of formula trees, only to find out that it can remove them immediately afterwards because they use the "wrong objects" in some unary inertia. For example, when instantiating the set of actions for the problem `strips-log-x-9` from the *logistics* domain used in the competition, 55088 actions are generated for which the instantiation procedure needs 527 seconds.

Consequently, there is the need for a further optimization of the instantiation process, which can be achieved through a separate treatment of unary inertia. The optimization, which is described in detail in this section, encodes all the unary inertia *obj*, *city*, *truck*, *airplane*, *location* and *airport* directly as types, which restrict the instantiation possibilities for the arguments of the operators. Running time for this example decreases to 63 seconds.[6]

We now give a precise notion of how implicit type information can be made explicit. First, for each unary inertia predicate p the new type symbol τ_p for *the type corresponding to* p is introduced.

Definition 12 *Let* p *be an inertia predicate of arity* 1*. The type* τ_p corresponding to p *is defined as the type whose domain comprises all constants* c *for which* $(p\ c)$ *holds in the initial state* $\mathcal{I}$*:*

$$dom(\tau_p) = \{c \mid (p\ c) \in \mathcal{I}\}$$

[6]The instantiation procedure implemented in IPP 3.3 that has been used in the competition is still a bit faster: It needs only 52 seconds for this example. However, this procedure uses a specialized algorithm which is only capable of handling *conjunctive* preconditions, and it generates a total of 62261 actions because no test for ground inertia is performed.

New types can be constructed from other types by intersecting or subtracting from each other the corresponding sets of constants.

Definition 13 *Let τ_1 and τ_2 be type names. Then $\tau_1 \cap \tau_2$ and $\tau_1 \setminus \tau_2$ are new type names. Their domains are defined as:*

$$dom(\tau_1 \cap \tau_2) = dom(\tau_1) \cap dom(\tau_2)$$

$$dom(\tau_1 \setminus \tau_2) = dom(\tau_1) \setminus dom(\tau_2)$$

After having extracted all types τ_p for unary inertia p from the initial state, the type structure of the domain representation is refined with the types τ_p and types that can be constructed from them.

Definition 14 *Let o be some operator and $?x$ be one of its parameters. Let p be a unary inertia. If $(p\ ?x)$ occurs in the preconditions of o or in the antecedent of one of its conditional effects, o is replaced by two new operators $o1$ and $o2$:*

- *In $o1$, the type τ that has been declared for $?x$ is restricted to $\tau \cap \tau_p$ and all occurrences of $(p\ ?x)$ are replaced with* TRUE.
- *In $o2$, the type τ that has been declared for $?x$ is restricted to $\tau \setminus \tau_p$ and all occurrences of $(p\ ?x)$ are replaced with* FALSE.

Similarly, quantified formulas in preconditions or antecedents of conditional effects are replaced.

Definition 15 *Let $\varphi = \forall\ ?x : \tau\ \psi$ be some universally quantified formula containing a unary inertia p with argument $?x$ of type τ. The formula φ is replaced with φ' defined as*

$$\varphi' = \forall\ ?x : \tau \cap \tau_p\ \ \psi[(p\ ?x)/\mathsf{TRUE}]\ \wedge\ \forall\ ?x : \tau \setminus \tau_p\ \ \psi[(p\ ?x)/\mathsf{FALSE}]$$

Let $\varphi = \exists\ ?x : \tau\ \psi$ be some existentially quantified formula containing a unary inertia p with argument $?x$. Then φ is replaced with φ'

$$\varphi' = \exists\ ?x : \tau \cap \tau_p\ \ \psi[(p\ ?x)/\mathsf{TRUE}]\ \vee\ \exists\ ?x : \tau \setminus \tau_p\ \ \psi[(p\ ?x)/\mathsf{FALSE}]$$

In the definition, $\psi[(p\ ?x)/\mathsf{TRUE}]$ and $\psi[(p\ ?x)/\mathsf{FALSE}]$ denote the formulas, which are obtained from ψ if all occurrences of $(p\ ?x)$ have been replaced with TRUE and FALSE, resp.

The soundness of the replacements follows from the observation that under the restriction $\tau \cap \tau_p$ the atomic formula $(p\ c)$ is always TRUE because only constants c are considered which are also in $dom(\tau_p)$. Under the restriction $\tau \setminus \tau_p$ only constants $c \in dom(\tau)$ that are *not* members of $dom(\tau_p)$ are considered and thus $(p\ c)$ is always FALSE.

We formally prove the soundness of the replacements for universally quantified formulas.

Theorem 3 (Soundness of Type Encodings)
Let p be a unary inertia predicate. Let $\varphi = \forall\ ?x : \tau\ \psi$ be a formula with $(p\ ?x)$ being a subformula of ψ. Let φ' be the formula φ gets replaced with according to definition 15. Then, for any state s that is reachable from the initial state holds

$$s \models \varphi \ \Leftrightarrow\ s \models \varphi'$$

Proof:
From the definition of τ_p we know that all constants $c \in \tau_p$ occur as arguments of p in the initial state, i.e., $\mathsf{N}(p\ c) = 1$. For those constants $c \notin \tau_p$, we have $\mathsf{N}(p\ c) = 0$. With Definition 6 and Theorem 1, we get for all states s that are reachable from the initial state:

$$(1)\ \ s \models (p\ c)\ \text{ for }\ c \in \tau_p$$

$$(2)\ \ s \not\models (p\ c)\ \text{ for }\ c \notin \tau_p$$

From this, we can immediately conclude for all states s that are reachable from the initial state:

$$(3)\ \ s \models \psi \Leftrightarrow s \models \psi[(p\ ?x)/\mathsf{TRUE}]\ \text{ for }\ c \in \tau_p$$

$$(4)\ \ s \models \psi \Leftrightarrow s \models \psi[(p\ ?x)/\mathsf{FALSE}]\ \text{ for }\ c \notin \tau_p$$

Thus, for any such state s

$$\begin{array}{lcl} s \models \forall\ ?x : \tau\ \psi & \Leftrightarrow & \text{for all } c \in \tau : s \models \psi[?x/c] \\ & \Leftrightarrow & \text{for all } c \in \tau \cap \tau_p : s \models \psi[?x/c] \text{ and} \\ & & \text{for all } c \in \tau \setminus \tau_p : s \models \psi[?x/c] \\ (3) \text{ and } (4) & \Leftrightarrow & \text{for all } c \in \tau \cap \tau_p : s \models \psi[(p\ ?x)/\mathsf{TRUE}] \text{ and} \\ & & \text{for all } c \in \tau \setminus \tau_p : s \models \psi[(p\ ?x)/\mathsf{FALSE}] \\ & \Leftrightarrow & s \models \forall\ ?x : \tau \cap \tau_p\ \psi[(p\ ?x)/\mathsf{TRUE}]\ \wedge\ \forall\ ?x : \tau \setminus \tau_p\ \psi[(p\ ?x)/\mathsf{FALSE}] \end{array}$$

As the last formula is exactly φ' as defined in Definition 15, the proposition follows. ■

The soundness of the replacement of operators follows from the fact that the modified operator set has the same set of ground instances that are generated by the instantiation procedure using inertia as described in Section 3. The soundness of the replacement of existentially quantified formulas follows with similar arguments as in the universally quantified case.

As an example, let us consider the operator from Figure 8 again. As there is no explicitly defined type for any of the three parameters they are assigned the default type *object*. When examining the first parameter *?obj*, IPP finds that it is used in the unary inertia predicate *obj*. Therefore, it generates two copies of the operator, restricts the parameter types according to Definition 14, and performs the corresponding atomic simplification of the unary inertia. The result is shown in Figure 9.

In the first operator, the atom TRUE can obviously be removed from the conjunction, which leads to a simplified precondition. As *all* constants in the STRIPS *logistics* problems are defined to be of type *object*, the domain $dom(object \cap \tau_{obj}) = dom(\tau_{obj})$ comprises exactly those constants c for which $(obj\ c)$ is contained in the initial state.

:action **load-truck**$^{(1)}$
:parameters *(?obj - object* $\cap\, \tau_{obj}$ *?truck ?loc)*
:precondition *(and (* TRUE *) (truck ?truck) (location ?loc)*
(at ?truck ?loc) (at ?obj ?loc))

:action **load-truck**$^{(2)}$
:parameters *(?obj - object* $\setminus \tau_{obj}$ *?truck ?loc)*
:precondition *(and (* FALSE *) (truck ?truck) (location ?loc)*
(at ?truck ?loc) (at ?obj ?loc))

Figure 9: Parameters and preconditions of the two new **load-truck** operators, which result from the encoding of the unary inertia predicate *obj* as a type.

In the second operator, the first atomic precondition has been replaced by FALSE as no constant in $dom(object \setminus \tau_{obj})$ can satisfy $(obj\ c)$. Thus, the whole precondition of this operator simplifies to FALSE and it can be removed from the operator set as it will never be applicable. Note that in the case of arbitrary first-order preconditions one cannot usually expect that operators can be removed immediately just after they have been generated.

Repeating this process for the other two parameters, always the second copy is removed immedeately after it has been generated and thus IPP obtains the final representation of the **load-truck** operator, which is shown in Figure 10.

:action **load-truck**
:parameters *(?obj - object* $\cap\, \tau_{obj}$
?truck - object $\cap\, \tau_{truck}$
?loc - object $\cap\, \tau_{location}$*)*
:precondition *(at ?truck ?loc) (at ?obj ?loc))*
:effect *(and (not (at ?obj ?loc)) (in ?obj ?truck))*

Figure 10: The new operator **load-truck**, which results from the encoding of all unary inertia as types and which replaces the original operator representation. This operator is identical with the one that is used in the typed version of this domain.

6 Empirical Results and Conclusion

Many examples could be presented, which nicely illustrate the benefits of an instantiation procedure that takes inertia into consideration. For example, in the *movie* domain used in the planning competition, 5 operators are declared to get snacks: *get-chips*, *get-dip*, *get-pop*, *get-cheese*, *get-crackers*. Each of them has a similar description, of which we only exemplify the *get-chips* operator:

get-chips
:parameters *(?x - chips)*
:precondition
:effect *(have-chips).*

One observes that the parameter $?x$ is not used anywhere in the operator description. If for example, 9 different constants are declared for each kind of snack, one obtains 9 ground instances of each operator, which are all identical and spam the search space of the planner. In all *movie* problems, the goals are reachable at time step 1, but a plan can only be extracted at time step 2, i.e., a permutation of all actions at time step 1 is performed by the complete search algorithm. Not very surprisingly, this takes almost 3 s in IPP 3.3 on a Sun Ultra 1/170 because 250973 actions must be tried before a solution is found. In contrast to this, when detecting the unused parameter, only one instance is generated for each operator, which dramatically reduces the search space down to 29 actions and thus a plan is found in only 0.06 s.

In the *assembly* domain, operators can be dramatically simplified because they contain so many inertia. For example, the complex precondition shown in Figure 1 uses 7 different predicates, but 5 of them are inertia. This means that each precondition must simplify to a formula only mentioning the fluents *incorporated* and *committed*. For many actions, the precondition reduces to a single atomic formula using only the *incorporated* predicate. IPP 4.0 is thus able to solve some *assembly* problems, while previously versions failed already during the instantiation, see Figure 11 for selected results.

problem name	actions	cpu seconds	search space	plan length
assem-x-1	114/760	1742.43	64 673 043	13/31
assem-x-2	84/882	18.03	848 829	13/30
assem-x-3	190/1248	0.83	108	10/33
assem-x-6	118/1800	46342.80	1 283 078 957	18/38

Figure 11: Performance of IPP on *assembly* problems on a Sun Ultra 1/170.

Column 2 shows the number of generated actions using the instantiation process with inertia compared to the number of all possible actions using naive enumeration. The search space is measured in the number of actions IPP tries until it finds a plan. The last column lists the number of time steps and the number of actions in the plan. Some other problems from this domain can be determined as unsolvable.

The determination of ground inertia helps IPP to discover information that it would not be able to find if only inertia *predicates* were analyzed. An interesting example of this behavior occurs in the *tower of Hanoi* domain. Given the operator

move*(?disc,?from,?to: disc)*
:precondition *(and (smaller ?to ?disc) (on ?disc ?from) (clear ?disc) (clear ?to))*
:effect *(and (clear(?from) (on ?disc ?to) (not (on ?disc ?from)) (not (clear ?to))*

which describes a legal move of discs, one notices that only a smaller disc can be moved onto a larger disc. IPP discovers that *smaller* is an inertia predicate and only generates the appropriate actions. But the action set also contains moves, which take a disc *from* a smaller disc and put it on a smaller disc. Indeed, the operator description says nothing about the relationship between the disc *?from* and the moving disc *?disc*, i.e., a move that takes a disc from a *smaller* disc and puts it on another smaller disc seems to be a legal action.

When performing the analysis of inertia on the ground level, IPP is able to find out that such moves are impossible. It detects that all instances of *(on ?disc ?from)*, where *?disc* is larger than *?from* are never made true by any action, i.e., they are positive inertia, and they do not hold in the initial state. Thus, these facts are irrelevant and all preconditions using them can be simplified to FALSE. Since all actions with FALSE as a precondition are removed from the action set, a further reduction of the size of the planning graph is achieved. For example, in the case of 3 discs, 10 out of 48 actions are eliminated. In the case of 8 discs, 140 out of 468 actions are removed. A search space of only 295.535 actions results and the plan of 255 steps is found in only 16 seconds.

The generation of the set of all ground actions for a given set of operators is a complex process which heavily influences the performance of any planner or pre-planning analysis method. The implementation comprises more than 5000 lines of C code, which are currently cleaned up and further improved and which will be made available to the planning community in the release of IPP 4.0. We hope that the instantiation procedure will become a useful part of reusable code that helps other researcher teams in setting up their own planners more quickly and without dealing with the burden of reimplementing the same preprocessing procedures again and again.

References

[Fox and Long, 1999] Fox, M. and Long, D. (1999). The detection and exploitation of symmetry in planning problems. Technical Report 1/99, Durham University.

[Geffner, 1999] Geffner, H. (1999). HSP: A heuristic search planner. web documentation.

[Kautz and Selman, 1996] Kautz, H. and Selman, B. (1996). Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the 14th National Conference of the American Association for Artificial Intelligence*, pages 1194–1201. AAAI Press.

[Koehler, 1998] Koehler, J. (1998). Solving complex planning tasks through extraction of subproblems. In Allen, J., editor, *Proceedings of the 4th International Conference on Artificial Intelligence Planning Systems*, pages 62–69. AAAI Press, Menlo Park.

[Koehler et al., 1997] Koehler, J., Nebel, B., Hoffmann, J., and Dimopoulos, Y. (1997). Extending planning graphs to an ADL subset. In [Steel, 1997], pages 273–285.

[McDermott, 1998] McDermott, D. (1998). Planning competition benchmark problems. web documentation, http://www.cs.yale.edu/users/mcdermott.hmtl.

[McDermott et al., 1998] McDermott, D. et al. (1998). *The PDDL Planning Domain Definition Language.* The AIPS-98 Planning Competition Comitee.

[Nebel et al., 1997] Nebel, B., Dimopoulos, Y., and Koehler, J. (1997). Ignoring irrelevant facts and operators in plan generation. In [Steel, 1997], pages 338–350.

[Pednault, 1989] Pednault, E. (1989). ADL: Exploring the middle ground between STRIPS and the Situation Calculus. In Brachman, R., Levesque, H., and Reiter, R., editors, *Proceedings of the 1st International Conference on Principles of Knowledge Representation and Reasoning*, pages 324–332, Toronto, Canada. Morgan Kaufmann.

[Steel, 1997] Steel, S., editor (1997). *Proceedings of the 4th European Conference on Planning*, volume 1348 of *LNAI*. Springer.