# An Iterative Algorithm for Synthesizing Invariants

**Jussi Rintanen**

Albert-Ludwigs-Universität Freiburg, Institut für Informatik
Georges-Köhler-Allee, 79110 Freiburg im Breisgau
Germany

## Abstract

We present a general algorithm for synthesizing state invariants that speed up automated planners and have other applications in reasoning about change. Invariants are facts that hold in all states that are reachable from an initial state by the application of a number of operators. In contrast to earlier work, we recognize the fact that establishing an invariant may require considering other invariants, and this in turn seems to require viewing synthesis of invariants as fixpoint computation. Also, the algorithm is not inherently restricted to invariants of particular syntactic forms.

## Introduction

For a given transition system, for example expressed as an initial state and a set of operators, invariants are facts that hold in all of its reachable states, or more precisely, they are true in the initial state, and their truth is preserved by the application of every operator (which is why they are called invariants.) Invariants can be applied in many kinds of planning algorithms for speeding them up. In algorithms based on backward chaining, like Graphplan (Blum and Furst 1997) and earlier partial-order planners, invariants rule out certain subgoals as unreachable. In algorithms that use neither regression nor progression and represent plan executions explicitly – for example the satisfiability planning approach – invariants extend the incomplete state descriptions and thereby reduce the amount of search needed (Kautz and Selman 1998; Gerevini and Schubert 1998). Invariants are useful also in many other kinds of planning algorithms that operate on partially described states.

Algorithms for computing invariants for automated planning have earlier been given by Kelleher and Cohn (1992), Rintanen (1998), Gerevini and Schubert (1998), and Fox and Long (1998). Kelleher and Cohn as well as Gerevini and Schubert verify that operators preserve the truth of an invariant on the basis of syntactic properties of the operators. Rintanen sketches an algorithm that computes 2-literal invariants from the ground instances of operators. Fox and Long obtain invariants as a byproduct of inferring types for operators.

In this paper we introduce a new algorithm for computing invariants. The algorithm is iterative like mutex computation in Graphplan (Blum and Furst 1997) and the algorithm by Rintanen (1998) (both of which use a ground representation of operators), operates on a schematic representation of operators, and generalizes earlier techniques. The algorithm is motivated by an inductive definition of invariants as formulae that are true in the initial state and are preserved by the application of every operator. Less general (and in restricted cases more efficient) algorithms can be obtained by specializing the general algorithm.

For schemata that represent 2-literal ground invariants we show that the algorithm is efficient. In this case – like with universally quantified invariants in general – the algorithm is strictly stronger than earlier algorithms combined. Invariants with more than two literals are often useful, but the conditions for inferring non-disjunctive facts (literals) from $n$-literal invariants for high $n$ are very strict because $n - 1$ atomic facts have to be inferred first, so short invariants seem to be the most important ones. Extensions like existential quantification and types can be handled within the algorithm by supplying new subprocedures to the main procedure. No changes in the main procedure are needed.

## Operators

An operator $p \Rightarrow e$ consists of a precondition $p$ and a postcondition $e$ that are sets of atomic literals. An operator can be applied if its preconditions are true, and as a result its postconditions become true. Many planning algorithms work with operators as described above but take input in schematic form; that is, a set of operators can be given as a schema from which each individual operator can be obtained by replacing the variables by constants.

In an operator schema $p \Rightarrow e$, the sets $p$ and $e$ consist of literals $a$ or $\neg a$ where $a$ are of the form $P(t_1, \ldots, t_n)$, $P$ is a predicate, and the terms $t_i$ are constants or variables. We sometimes write $P(T)$ where $T$ is a sequence of terms. The ground literals represented by a literal schema are obtained by replacing variables with constants in all possible ways. For simplicity of presentation we assume that all variables have the same type and that different variables are instantiated with different constants. The latter assumption is relevant only in the main procedure of the algorithm.

## Form of Invariants

The invariant schemata we consider are of the form $(x_1 \neq x_1' \wedge \cdots \wedge x_n \neq x_n') \rightarrow (L_1 \vee \cdots \vee L_m)$, where $L_i$ are schematic literals $P_i(x_{1,1}, \ldots, x_{1,n_1})$ or $\neg P_i(x_{1,1}, \ldots, x_{1,n_1})$, and $x_i$ and $x_i'$ are variables. All variables and predicate symbols may be different.

Each invariant schema corresponds to a set of ground invariants that are obtained by replacing the variables by constants without violating the inequalities.

## Synthesis of Invariants

We present an iterative algorithm for computing invariants from schematic representations of operators. The algorithm produces a sequence $\Sigma_0, \Sigma_1, \ldots, \Sigma_n, \Sigma_{n+1}$ of sets of schemata such that $\Sigma_n = \Sigma_{n+1}$, $\Sigma_0$ is satisfied by the initial state, and each $\Sigma_i$ is obtained from $\Sigma_{i-1}$ by identifying candidate invariants that may be falsified by operators that are applicable in states that satisfy $\Sigma_{i-1}$, and replacing them by weaker candidate invariants. The set $\Sigma_n$ that is preserved by all operator applications consists of invariants.

Because exact descriptions $\Sigma_i$ of states reachable with $i$ steps or fewer may be of exponential size and computing invariants is PSPACE-hard ($\sigma$ is an invariant iff there is no plan that achieves $\neg\sigma$), syntactic restrictions on $\Sigma_i$ have to be considered. We do not allow constant symbols in the invariants and have an upper bound on clause length. As a consequence, the sets $\Sigma_i$ are only an upper bound on the reachable states. These restrictions also guarantee a polynomial upper bound on the runtime.

The main procedure of the algorithm is given in Figure 1. The functions extend$(p, \Sigma)$, update$(p, e)$, preserves$(e, u, \sigma)$ and weaken$(\sigma)$ are described in the following sections. The algorithm first identifies candidate invariants $\Sigma_0$, the ground instances of which are true in the initial state. The computation starts from all the atomic schemata with predicates $P_i$ that occur in the problem instance. Here $X_i$ are sequences of distinct variables. Then the algorithm goes through stages $i = 1, 2, \ldots$, considering each operator $o \in O$ at each stage. The ground operator $p \Rightarrow e$ is obtained from $o$ by replacing variables occurring in $o$ by new distinct constant symbols. The function call extend$(p, \Sigma_{i-1})$ extends the grounded precondition by using the candidate invariants $\Sigma_{i-1}$ identified at the previous stage. If an inconsistent set is obtained (containing the empty clause), the precondition was not consistent with $\Sigma_{i-1}$ and the operator is not applicable. For applicable operators a description of the possible successor states is obtained with the function call update$(p', e)$, and the preservation of each candidate invariant is tested against it. If a candidate invariant cannot be shown to be preserved by an operator, it is replaced by weaker candidate invariants.

In the following sections we describe the auxiliary functions of the algorithm. A familiarity with notions like unification, substitutions and so on is assumed. When we write about unifiers, we mean most general unifiers. Because the algorithm is for efficiency reasons incomplete, there is a certain freedom in implementing the auxiliary functions. We describe the requirements the functions have to satisfy for

INPUT: an initial state $I$, a set $O$ of operators
OUTPUT: a set of invariants for $I, O$

$\Sigma_0 := \{P_1(X_1), \neg P_1(X_1), P_2(X_2), \ldots\}$
**WHILE** there is $\sigma \in \Sigma_0$ that is false in $I$ **DO**
  $\Sigma_0 := (\Sigma_0 \backslash \{\sigma\}) \cup$ weaken$(\sigma)$;
i := 0;
**REPEAT**
  $i := i + 1$;
  $\Sigma_i := \Sigma_{i-1}$;
  **FOR EACH** $o \in O$ **DO**
    let $p \Rightarrow e$ be a ground instance of $o$;
    $p' :=$ extend$(p, \Sigma_{i-1})$;
    **IF** $p'$ is consistent
    **THEN**
      $u :=$ update$(p', e)$;
      **WHILE** not preserves$(e, u, \sigma)$ for some $\sigma \in \Sigma_i$
      **DO** $\Sigma_i := (\Sigma_i \backslash \{\sigma\}) \cup$ weaken$(\sigma)$;
**UNTIL** $\Sigma_i = \Sigma_{i-1}$;
**RETURN** $\Sigma_i$;

Figure 1: The main procedure of the algorithm

the algorithm to be correct, and outline one possible implementation.

### The Function extend$(p, \Sigma)$

To see which facts are true after an operator is applied, we need to know which facts are true before the operator is applied. Obviously, the preconditions $p$ of the operator are true, but assuming that certain facts $\Sigma$ hold, we can infer the truth of several other facts as well. The function extend$(p, \Sigma)$ performs these inferences.

*For the correctness of the algorithm the function has to satisfy $p \cup \Sigma \models$ extend$(p, \Sigma)$.*

The function extends a set of ground literals $p$ by applying the resolution rule between clauses from $\Sigma$ and $p$. Resolving clauses in $\Sigma$ with each other would produce clauses already in $\Sigma$ or (for clauses with 3 or more literals) longer clauses that would often violate the upper bound on clause length. Notice that not doing all possible inferences does not sacrifice the correctness of the algorithm.

So for $E \rightarrow (A_1 \vee \cdots \vee A_m) \in \Sigma$ choose $n \in \{m - 1, m\}$, $\{l_1, \ldots, l_n\} \subseteq p$, and $L_1 = \{A_{i_1}, \ldots, A_{i_n}\} \subseteq L_2 = \{A_1, \ldots, A_m\}$. Then for every $j \in \{1, \ldots, n\}$ unify $\overline{l_j}$ with $A_{i_j}$ to obtain a unifier $\theta$ (inequalities $E$ may not be violated.) Now the clause $(E \rightarrow L)\theta$ for $L = L_2 \backslash L_1$ can be inferred. If $n = m$ we get the empty clause.

**Example 1** Consider an operator that moves a block from the top of a block on top of another block. This operator has a ground instance with the precondition $p = \{on(A, B), clear(A), clear(C)\}$. Let

$$\Sigma = \{x \neq y \rightarrow (\neg on(x, z) \vee \neg on(y, z)),$$
$$x \neq y \rightarrow (\neg on(z, x) \vee \neg on(z, y)),$$
$$\neg clear(x) \vee \neg on(y, x)\}.$$

Now

$$\begin{aligned}
\text{extend}(p, \Sigma) \;=\; \{ & on(A,B), clear(A), clear(C), \\
& x \neq A \to \neg on(x,B), \\
& x \neq B \to \neg on(A,x), \\
& A \neq y \to \neg on(y,B), \\
& B \neq y \to \neg on(A,y), \\
& \neg on(x,A), \neg on(x,C), \neg clear(B) \}.
\end{aligned}$$

This is because for example the literal $on(y,z)$ in the clause $x \neq y \to (\neg on(x,z) \vee \neg on(y,z))$ unifies with $on(A,B)$ and hence produces $x \neq A \to \neg on(x,B)$. ∎

## The Function update$(p, e)$

Given an incomplete description $p$ of a state in which a ground operator with the postcondition $e$ is applied, the function update$(p, e)$ computes an incomplete description of the resulting state. This involves modifying members of $p$ according to the ground literals in $e$ that become true. The set $p$ consists of schemata $\sigma = E \to \phi$ where $E$ is a conjunction of inequalities $x \neq y$ and $\phi$ is a disjunction of literals. $E$ may be the empty conjunction, which is defined to be true.

*For the correctness of the algorithm the function must satisfy the following. If an operator making the atomic literals $e$ true is applied in a state satisfying $p$, then the successor state satisfies update$(p, e)$.*

Updating members of $p$ according to the literals in $e$ can be done one at a time, and separately with respect to every member of $e$. Positive and negative literals are treated symmetrically, so we consider only positive ground literals $P(c_1, \ldots, c_n) \in e$. We consider every member $\sigma$ of $p$ in turn, and show how it has to be modified to reflect the update according to $P(c_1, \ldots, c_n)$.

1. If $P$ does not occur in $\sigma$, $\sigma$ is left intact.

2. If $\phi$ consists of more than one literal, delete $\sigma$.[1]

3. If $\sigma$ is $\neg P(c_1, \ldots, c_n)$, delete $\sigma$.

4. If $\sigma$ is $E \to \neg P(t_1, \ldots, t_n)$, then unify $P(c_1, \ldots, c_n)$ with $P(t_1, \ldots, t_n)$. If unification succeeds with the unifier $\theta$ and the inequalities $E\theta$ are satisfied, the following changes are made. Let $c_{i_1}/x_{i_1}, \ldots, c_{i_m}/x_{i_m}$ be the unifier. Now $\sigma$ is replaced by $m$ candidate invariants $(c_{i_j} \neq x_{i_j} \wedge E) \to \neg P(t_1, \ldots, t_n)$ for $j \in \{1, \ldots, m\}$.

   In many cases, like in the example below, at most one variable unifies with a constant and hence $m = 1$.

5. If $\sigma$ is $E \to P(t_1, \ldots, t_n)$ and $P(c_1, \ldots, c_n)$ unifies with $P(t_1, \ldots, t_n)$ producing a one-element unifier $\theta = c/x$ and $x$ is the only variable occurring in $\sigma$, remove $c \neq x$ from $E$ (if it is in it).[2]

---

[1] The function would be stronger if it retained $\sigma$ when none of the literals in the ground instances of $\sigma$ are affected. However, our implementation of *extend* produces clauses with one literal only, and hence the change would not make a difference.

[2] A stronger implementation of this case – or alternatively of the function *preserves* – may sometimes be necessary for obtaining more invariants. Now there is no means for combining schemata that get split in (4). The splitting is essentially a way of handling disjunctive antecedents.

6. If in the resulting set no literal unifies with $P(c_1, \ldots, c_n)$, $P(c_1, \ldots, c_n)$ is added to the set.

**Example 2** Let

$$\begin{aligned}
p \;=\; \{ & on(A,B), clear(A), clear(C), \\
& x \neq A \to \neg on(x,B), \\
& x \neq B \to \neg on(A,x), \\
& \neg on(x,A), \neg on(x,C), \neg clear(B) \}.
\end{aligned}$$

We make the following ground literals true in $p$.

$$e = \{ on(A,C), \neg on(A,B), clear(B), \neg clear(C) \}$$

The result is the following.

$$\begin{aligned}
\text{update}(p, e) \;=\; \{ & clear(A), \neg clear(C), \neg on(x,B), \\
& x \neq C \to \neg on(A,x), \\
& x \neq A \to \neg on(x,C), \\
& \neg on(x,A), on(A,C), clear(B) \}
\end{aligned}$$

For example, $x \neq B \to \neg on(A,x)$ is transformed to $x \neq C \wedge x \neq B \to \neg on(A,x)$ by $on(A,C)$, and then to $x \neq C \to \neg on(A,x)$ by $\neg on(A,B)$. ∎

## The Function preserves$(e, u, \sigma)$

When making the ground literals $e$ true in some state so that a state described by $u$ is reached, we check whether the truth of the instances of $\sigma$ is preserved.

*For the correctness of the algorithm the function preserves$(e, u, \sigma)$ may return true only if $u \models \sigma$ or $e$ does not falsify any literal in any ground instance of $\sigma$.*

The function first tests whether a literal in a ground instance of $\sigma$ is falsified when the literals in $e$ become true. If not, the function returns *true*. Otherwise, we unify complements of literals in $e$ with literals in $\sigma$ in all possible ways. If $\sigma\theta$ is true in $u$ for all unifiers $\theta$ we return *true*. Otherwise we return *false*.

The truth of $\sigma\theta$ in $u$ is tested by a function that tests whether all ground instances of the first are entailed by ground instances of the second. Note that the test does not have to be complete for the whole algorithm to be correct, and an incomplete test suffices. We implement it as testing $\sigma' \models \sigma\theta$ for universally quantified first-order clauses (equivalently: refutability of $\sigma' \wedge \neg\sigma\theta$), where $\sigma' \in u$. This is by applications of unit resolution.

**Example 3** Let

$$\begin{aligned}
e \;=\;& \{ on(A,C), \neg on(A,B), clear(B), \neg clear(C) \}, \\
u \;=\;& \{ clear(A), \neg clear(C), \\
& x \neq C \to \neg on(A,x), \\
& x \neq A \to \neg on(x,C), \\
& \neg on(x,A), on(A,C), clear(B) \}, \text{ and} \\
\sigma \;=\;& y \neq z \to (\neg on(x,y) \vee \neg on(x,z)).
\end{aligned}$$

The complement of the disjunct $\neg on(x,y)$ in $\sigma$ unifies with $on(A,C)$. The unifier $\theta$ assigns $x = A$ and $y = C$. To see whether the truth of $\sigma$ is preserved when $e$ is made true, we have to check whether $(y \neq z \to \neg on(x,z))\theta = C \neq z \to \neg on(A,z)$ is included in $u$. It is, because $x \neq C \to \neg on(A,x) \in u$ has exactly the same ground instances. ∎

## The Function weaken($\sigma$)

When it cannot be shown that a candidate invariant is preserved by an operator, it is rejected. There may, however, be closely related invariants that are true in all reachable states and hence preserved by all operators. So when a candidate invariant is rejected, we produce a number of new ones that are weaker in the sense that they hold in more states.

*For the termination of the algorithm the schemata $\sigma' \in$ weaken($\sigma$) have to satisfy $\sigma \models \sigma'$ and $\sigma' \not\models \sigma$.*

We have three weakening operations: adding a disjunct to the consequent, adding a conjunct $x \neq y$ to the antecedent, and identifying two variables by replacing occurrences of one by the other. As discussed earlier, the computation of arbitrarily complex invariants is not feasible. Hence falsified candidate invariants with a certain number of literals in the consequent or in the antecedent are not weakened, but completely ignored.

**Example 4** Consider the schema $\sigma = x \neq y \rightarrow P(x,y) \vee Q(y,z)$. Let $P$ and $Q$ be the only predicates. By adding a new literal we obtain the following four weaker schemata.

$$x \neq y \rightarrow P(x,y) \vee Q(y,z) \vee P(u,v)$$
$$x \neq y \rightarrow P(x,y) \vee Q(y,z) \vee Q(u,v)$$
$$x \neq y \rightarrow P(x,y) \vee Q(y,z) \vee \neg P(u,v)$$
$$x \neq y \rightarrow P(x,y) \vee Q(y,z) \vee \neg Q(u,v)$$

By adding a new inequality we obtain the following.

$$x \neq y \wedge y \neq z \rightarrow P(x,y) \vee Q(y,z)$$
$$x \neq y \wedge x \neq z \rightarrow P(x,y) \vee Q(y,z)$$

By identifying two variables we obtain the following.

$$x \neq y \rightarrow P(x,y) \vee Q(y,x)$$
$$x \neq y \rightarrow P(x,y) \vee Q(y,y)$$

Now weaken($\sigma$) consists of the above schemata. ∎

## An Example

Consider the blocks world with blocks A, B and C, and the initial state where A and B are on the table, and C is on top of B. The algorithm starts with `on(x,y)`, `ontable(x)`, `clear(x)`, `-on(x,y)`, `-ontable(x)`, and `-clear(x)` and weakens them with the initial state until the following 2-literal schemata $\Sigma_0$ are obtained.

```
1.  (x!=z)=>(-on(z,u)|-on(x,u))
2.  (y!=u)=>(-on(z,u)|-on(z,y))
3.  -on(y,y)
4.  -on(z,u)|-on(u,y)
5.  -on(y,z)|-ontable(y)
6.  -clear(z)|-on(x,z)
7.  clear(z)|-on(z,y)
8.  ontable(z)|-on(x,z)
9.  ontable(y)|clear(y)
```

From `-on(z,u)|-on(u,y)` an invariant is later obtained by identifying $z$ and $y$. Also the last three are not invariants. These schemata essentially say that all stacks of blocks are of height 2.

For producing $\Sigma_i$ for $i \geq 1$ the operators are considered. If it cannot be shown that a candidate invariant is preserved by the application of an operator, it must be rejected. This happens to schemata 4, 7, 8, 9. The last three do not yield weaker invariants because of the restriction to 2-literal clauses. The first iteration produces the following $\Sigma_1$, and at the second iteration we see that $\Sigma_1$ is the fixpoint ($\Sigma_1 = \Sigma_2$.)

```
1.  (x!=z)=>(-on(z,u)|-on(x,u))
2.  (y!=u)=>(-on(z,u)|-on(z,y))
3.  -on(y,y)
4.  -on(z,u)|-on(u,z)
5.  -on(y,z)|-ontable(y)
6.  -clear(z)|-on(x,z)
```

## Soundness

Proof of soundness of invariant computation is by induction on the number of iterations. States that are reachable from the initial state with $i$ consecutive operations or less satisfy all ground instances of $\Sigma_i$.

The base case is directly because the first step of the algorithm ensures that schemata in $\Sigma_0$ have only ground instances that are true in the initial state.

For the inductive case we have to show that for a state $s$ reachable with $i$ operations there are no candidate invariants $\sigma \in \Sigma_i$ that are false in $s$. So assume a candidate invariant $\sigma$ is false in a state that is reachable from the initial state by $i$ consecutive operations. Hence there is a ground instance $p \Rightarrow e$ of an operator that is applicable after $i - 1$ consecutive operations from the initial state and that makes a ground instance of $\sigma$ false. By the induction hypothesis $\Sigma_{i-1}$ does not falsify $p$. Therefore $p' = \text{extend}(p, e)$ is consistent. The schemata in update$(p', e)$ are true in the state that is reached by applying the operator. And preserves$(e, \text{update}(p', e), \sigma)$ returns false. The last three facts are directly the correctness criteria the functions satisfy. Therefore $\sigma \notin \Sigma_i$.

The algorithm terminates because the number of states satisfying the candidate invariants increases at each iteration, and as there are finitely many atomic facts, there is an upper bound on the number of states.

## Computational Complexity

Given a fixed upper bound on the number of literals in the invariants, the number of candidate invariants is polynomial. All the auxiliary functions run in polynomial time. The number of iterations is bounded by the number of candidate invariants. Hence the algorithm runs in polynomial time.

Interestingly, like shown in the outline of the soundness proof above, the number of iterations is also bounded by the longest of the shortest paths from the initial state to a reachable state. However, like shown by the experiments in the next section, the number of iterations is much lower in practice. This is because the candidate invariants cannot exactly describe all sets of states (assuming that there are no constant symbols and clause length is bounded) and therefore sets $\Sigma_i$ often represent much larger sets of states than those reachable with $i$ steps. The iteration therefore terminates much faster than what the theoretical upper bounds predict.

| domain | ops | 2-literal invars | time | 3-literal invars | time |
|---|---|---|---|---|---|
| bw-large.a/p | 3 | 6 | 0.43 | 7 | 44.19 |
| bw-large.d/p | 3 | 6 | 1.18 | 7 | 105.76 |
| logistics.a | 6 | 4 | 1.28 | 4 | 160.09 |
| logistics.d | 6 | 4 | 1.68 | 4 | 183.92 |
| hanoi.6 | 1 | 6 | 0.45 | 7 | 53.41 |
| hanoi.15 | 1 | 6 | 1.15 | 7 | 259.82 |

Table 1: Runtimes of invariant synthesis in seconds. The numbers of iterations when computing 2-literal invariants for bw-large, logistics and hanoi were respectively 2, 6 and 1, and for 3-literal invariants respectively 3, 11 and 3.

## Experiments

We have implemented the algorithm, including typed variables, and tried it on a number of benchmarks: the well-known blocks world, logistics and towers of Hanoi.

Data from a number of runs are given in Table 1. The runs were on a 360 MHz Sun Ultra workstation. The program is compiled Standard ML. We give runtimes for the generation of 2-literal invariants (the only important form in many applications) and for comparison also for 3-literal invariants. Only one inequality was allowed in the antecedents. Most of the time is spent in weakening the candidate invariants with the initial state. Main sources of computational overhead are the generation of many candidate invariants by the function *weaken*, almost all of which are later rejected, and the identification of redundancies by testing inclusion between candidate invariants. More sophisticated implementation techniques would reduce these overheads substantially, especially in the 3-literal case.

Not all domains have interesting $n$-literal invariants for any given $n$. For the blocks world with $n$ blocks there are $m$-literal invariants (for $m \leq n$) stating that the *on* relation is acyclic. The only 3-literal invariant for the blocks world (and towers of Hanoi) that is not a consequence of a 2-literal invariant is `-on(x,y)|-on(y,z)|-on(z,x)`. The logistics domain does not have any.

Invariants inferred by our algorithm for common planning benchmarks are given next. The 2-literal invariants for the blocks world were given earlier. For towers of Hanoi we get the following invariants.

```
1. -on(x:DISK,x:DISK)
2. -on(y:DISK,z:DISK)|-on(z:DISK,y:DISK)
3. -on(x:DISK,y:DISK)|-free(y:DISK)
4. (x!=z)=>(-at(y:DISK,z:PEG)|-at(y:DISK,x:PEG))
5. (x!=z)=>(-on(y:DISK,z:DISK)|-on(y:DISK,x:DISK))
6. (x!=y)=>(-on(y:DISK,z:DISK)|-on(x:DISK,z:DISK))
```

And for the logistics domain the following.

```
1. (y!=u)=>(-at(z:{PACKAGE,TRUCK},u:{PORT,AIRPORT})
           |-at(z:{PACKAGE,TRUCK},y:{PORT,AIRPORT}))
2. -at(z:PACKAGE,u:{PORT,AIRPORT})
   |-transport(z:PACKAGE,y:{TRUCK,AIRPLANE})
3. (y!=u)=>(-at(z:{PACKAGE,AIRPLANE},u:AIRPORT)
           |-at(z:{PACKAGE,AIRPLANE},y:AIRPORT))
4. (y!=u)=>(-transport(z:PACKAGE,u:{TRUCK,AIRPLANE})
           |-transport(z:PACKAGE,y:{TRUCK,AIRPLANE}))
```

Invariants 1 and 3 overlap. This is because airplanes cannot be at a port, and therefore objects at airports and objects at airports or ports get handled separately.

## Related Work

Derivation of invariants from first-order formalizations of actions has been investigated by Zhang and Foo (1997). They give a general rule that is based on inferring which fluents are preserved by an action and that corresponds to the computation performed by our function *update*. They also give derivations of many blocks world invariants.

Gerevini and Schubert's (1998) techniques for computing invariants appear to be more general than those by Kelleher and Cohn (1992). Their method for computing implicational invariants is a special case of the computation performed by our functions *update* and *preserves*: a disjunction is an invariant if for every operator, neither disjunct is falsified by the operator, or one disjunct is made true, or one disjunct is a precondition and it is not made false by the operator. For inferring that in the blocks world there can be at most one block on top of a block and that a block with another block on top of it is not clear, Gerevini and Schubert propose techniques that are special cases of our idea of strengthening operator preconditions with candidate invariants. Gerevini and Schubert say that for the blocks world they cannot infer that a block is on top of at most one block or that two blocks cannot simultaneously be on top of each other.

Fox and Long (1998) address the problem of inferring types for objects on the basis of the operators and an initial state. Data obtained in that computation can be used in inferring invariants that state that objects have exactly one of several (positive) properties (or, in some cases, at most $n$ if the initial state had $n$.) Many invariants are not recognized by Fox and Long, like our blocks world invariants 1, 3, 4 and 6, and the following.

**Example 5** Consider the operators $B(x) \Rightarrow \neg A(x)$, $A(x) \Rightarrow \neg B(x)$, $\Rightarrow A(x)$, and $\Rightarrow B(x)$, and an initial state in which for every $x$ at least one of $A(x)$ and $B(x)$ is true. Clearly $A(x) \vee B(x)$ is an invariant. ∎

For example for the logistics domain Fox and Long infer $\forall x \exists y \text{at}(x, y)$ for vehicles $x$. Our algorithm as described above does not use existential variables, and hence does not infer this invariant. However, Fox and Long's algorithm is strictly weaker than the obvious extension of our algorithm to existential variables that is pointed out in the conclusions.

Invariants cannot in general be produced separately. Interaction between invariants is often essential in establishing them, and the invariants in the following example cannot be identified with the techniques proposed earlier (Kelleher and Cohn 1992; Gerevini and Schubert 1998; Fox and Long 1998), but our algorithm finds them immediately.

**Example 6** Consider the operators $\neg C(x) \Rightarrow \neg A(x)$, $\neg A(x) \Rightarrow \neg C(x)$, and $A(x) \wedge C(x) \Rightarrow \neg B(x)$ and an initial state that satisfies $I = \{A(x) \vee B(x), B(x) \vee C(x)\}$. The

formulae $I$ are invariants for the initial state and the operators, but verifying that for example $A(x) \vee B(x)$ is preserved by $\neg C(x) \Rightarrow \neg A(x)$ requires extending the precondition $\neg C(x)$ to $B(x), \neg C(x)$ by the invariant $B(x) \vee C(x)$. ∎

The problem of testing whether given formulae are invariants of a transition system has been extensively investigated in the context of computer-aided verification (Bensalem, Lakhnech, and Saidi 1996).

## Conclusions

We have presented an algorithm for computing invariants for automated planning. The main differences to earlier techniques are that the algorithm is not restricted to invariants of particular syntactic forms, it works uniformly for all invariants, and it is formalized as the iterative computation of a fixpoint. Earlier techniques establish each invariant separately and fail to produce invariants that our algorithm produces. Fixpoint computation is needed because interdependencies between invariants may be complex, and it is not in general possible to infer some invariants first and then use them for inferring others. Example 6 shows how two invariants can depend on each other and have to be established in parallel.

We have been able to show that our algorithm can be implemented efficiently for the practically most important case of 2-literal invariants. Improved implementation techniques may make it practical for $n$-literal invariants for $n \geq 3$.

Our algorithm is not restricted to computing invariants for only one initial state. Given a set $S$ of initial states represented as $\Sigma_0$ such that $S \models \Sigma_0$, we start the computation from this $\Sigma_0$ instead of the schemata satisfied by $I$. Similarly, the algorithm can be used for testing whether given (non-automatically identified) schemata $\Sigma_0$ are invariants.

This work can be extended to several directions. As an alternative to the formula-based inexact representations of reachable states used in the current paper, standard techniques from symbolic model-checking (Burch et al. 1994) that use binary decision diagrams could be used for performing an exact reachability analysis. Extracting invariants from the resulting binary decision diagrams is straightforward. The main problem in this approach is the size of the binary decision diagrams on bigger problems. Also, only ground invariants could be extracted.

In this paper we only consider schemata with universal variables that represent conjunctions of ground clauses. Schemata with existential variables may represent arbitrarily long disjunctions of ground literals, for example $\forall x(ontable(x) \vee \exists y on(x, y))$ for the blocks world. For handling existential variables one only needs to extend the four auxiliary functions, which is straightforward. Another extension that is not described in the paper is typed variables. Also this extension is an easy exercise.

## Acknowledgements

We thank the reviewers for many valuable comments.

## References

Bensalem, S.; Lakhnech, Y.; and Saidi, H. 1996. Powerful techniques for the automatic generation of invariants. In Alur, R., and Henzinger, T. A., eds., *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102 of *Lecture Notes in Computer Science*, 323–335. New Brunswick, New Jersey, USA: Springer-Verlag.

Blum, A. L., and Furst, M. L. 1997. Fast planning through planning graph analysis. *Artificial Intelligence* 90(1-2):281–300.

Burch, J. R.; Clarke, E. M.; Long, D. E.; MacMillan, K. L.; and Dill, D. L. 1994. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 13(4):401–424.

Fox, M., and Long, D. 1998. The automatic inference of state invariants in TIM. *Journal of Artificial Intelligence Research* 9:367–421.

Gerevini, A., and Schubert, L. 1998. Inferring state constraints for domain-independent planning. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98) and the 10th Conference on Innovative Applications of Artificial Intelligence (IAAI-98)*, 905–912. AAAI Press.

Kautz, H., and Selman, B. 1998. The role of domain-specific knowledge in the planning as satisfiability framework. In Simmons, R.; Veloso, M.; and Smith, S., eds., *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems*, 181–189.

Kelleher, G., and Cohn, A. G. 1992. Automatically synthesising domain constraints from operator descriptions. In Neumann, B., ed., *Proceedings of the 10th European Conference on Artificial Intelligence*, 653–655. John Wiley & Sons.

Rintanen, J. 1998. A planning algorithm not based on directional search. In Cohn, A. G.; Schubert, L. K.; and Shapiro, S. C., eds., *Principles of Knowledge Representation and Reasoning: Proceedings of the Sixth International Conference (KR '98)*, 617–624. Morgan Kaufmann Publishers.

Zhang, Y., and Foo, N. Y. 1997. Deriving invariants and constraints from action theories. *Fundamenta Informaticae* 30(1):109–123.