# Inferring State Constraints for Domain-Independent Planning*

**Alfonso Gerevini**
Dipartimento di Elettronica per l'Automazione
Università di Brescia
Via Branze 38, 25123 Brescia, Italy
E-mail: gerevini@bsing.ing.unibs.it

**Lenhart Schubert**
Department of Computer Science
University of Rochester
Rochester, NY 14627-0226
E-mail: schubert@cs.rochester.edu

**Abstract**

We describe some new preprocessing techniques that enable faster domain-independent planning. The first set of techniques is aimed at inferring state constraints from the structure of planning operators and the initial state. Our methods consist of generating hypothetical state constraints by inspection of operator effects and preconditions, and checking each hypothesis against all operators and the initial conditions. Another technique extracts (supersets of) predicate domains from sets of ground literals obtained by Graphplan-like forward propagation from the initial state. Our various techniques are implemented in a package called DISCOPLAN. We show preliminary results on the effectiveness of adding computed state constraints and predicate domains to the specification of problems for SAT-based planners such as SATPLAN or MEDIC. The results suggest that large speedups in planning can be obtained by such automated methods, potentially obviating the need for adding hand-coded state constraints.

## Introduction

State constraints in a planning domain are laws that hold in every state that can be reached from the initial state. For instance in blocks world problems any object $y$ other than the TABLE is necessarily NOT CLEAR in a state where another object $x$ is ON $y$. State constraints are important in planning for two reasons. In planning formalisms that specify only the direct (or "primary") effects of actions, state constraints are needed to infer indirect effects (i.e., the ramifications of the direct effects). In all types of formalisms, state constraints can also be a powerful means of restricting the search space within which a plan is sought.

It is this latter role of state constraints that concerns us here. We are particularly interested in recent planners that use propositional satisfiability testing to discover plans. These are currently among the most powerful domain-independent planners, but their performance appears to depend strongly on whether or not state constraints are included in the propositional formulation of a planning problem (even when these constraints are logically unnecessary for the solution of the problem). This is true, for instance, of Kautz *et al.*'s SATPLAN system, and, as we shall see, of Ernst *et al.*'s method of compiling UCPOP domain specifications and a particular problem into SAT clauses and then applying a solver such as Tableau (Crawford & Auton 1993), Walksat (Selman, Kautz, & Cohen 1994) or Relsat (Bayardo & Schrag 1997) to obtain a plan.

To make effective use of state constraints and predicate domain constraints in the above types of planners, we need to know what these constraints are. But many planning formalisms leave some or all such constraints implicit in the structure of the operators and the initial state. In the following we describe several ways of inferring state constraints from the structure of STRIPS-like operators. In addition to the kinds of state constraints traditionally discussed in planning, we also consider constraints on predicate domains, in the form of sets of possible tuples of arguments. We infer such sets by forward propagation of ground atoms from the initial state in the manner of Graphplan (Blum & Furst, 1995) but without exclusion calculations. These methods have been implemented in our DISCOPLAN package (DIScovering COnstraints for PLANning), and we propose some additional, provably correct techniques to be added in the future.

We then report preliminary results in the use of both types of constraints to accelerate SAT-based planners (without recourse to any hand-coded constraints). These results suggest that automated methods for inferring constraints in domain-independent planning should be able to largely supplant hand-coding of such constraints, providing comparable speedups in planning.

## Inferring State Constraints from Domain Descriptions

Let us treat state (or situation or time) variables as implicit, as is done in planners such as STRIPS or UCPOP. Syntactically speaking, a state constraint in a particular planning domain, for particular initial conditions, could then be any closed logical formula involving the predicates and individual constants occurring in the domain. (Some planners allow functions as well, but we shall neglect these.) As a state constraint, such a formula is regarded as holding in all states; we could make this quantification over states (or times) explicit by adding a state (or time) variable $s$ to each fluent predicate, and prefixing a wide-scope universal quantifer $(\forall s)$ to the formula. Examples of such state constraints in a blocks world, with implicit universal quantification of free variables, are

$\text{ON}(x,y) \wedge \text{NEQ}(y,\text{TABLE}) \Rightarrow \neg\text{CLEAR}(y)$,

$\text{ON}(x,y) \wedge \text{NEQ}(y,z) \Rightarrow \neg\text{ON}(x,z)$.

Note that in the first constraint, all variables occurring anywhere also occur in one of the antecedent literals, namely $\text{ON}(x,y)$. We will refer to such state constraints as *implicative constraints*. The second constraint (though also formulated as an implication here) contains no literal with an all-inclusive set of variables. It is a *single-valuedness constraint* (*sv-constraint*), i.e., it asserts that no object $x$ can be ON more than one object $y$. It is our impression from examining various domains such as those in the UCPOP test suite that the most prevalent constraints – at least among those expressible as simple logical formulas – are implicative and sv-constraints. We will abbreviate sv-constraints by writing "*" or a variable prefixed by "*" for the value that must be unique (if it exists at all), for any given values of the remaining arguments. So the second formula above becomes $\text{ON}(x,*)$.

Frequently there are also constraints on co-occurrence of predicate arguments. For instance, in a blocks world with just two blocks A, B, the possible ON-relationships might be ON(A,TABLE), ON(A,B), ON(B,TABLE), and ON(B,A). Although we use an entirely separate technique for inferring such predicate domains, note that they *do* comprise state constraints. For instance, the constraint above can be written as

$$\text{ON}(x,y) \Rightarrow (x=\text{A} \wedge y=\text{TABLE}) \vee (x=\text{A} \wedge y=\text{B}) \vee (x=\text{B} \wedge y=\text{TABLE}) \vee (x=\text{B} \wedge y=\text{A}).$$

We now describe our specific techniques for inferring implicative and sv-constraints, and then our techniques for inferring predicate domains. While many useful implicative constraints can be inferred in isolation, others can only be inferred jointly with certain sv-constraints. This is true, for example, in the case of the constraint $\text{ON}(x,y) \Rightarrow \neg\text{CLEAR}(y)$ (possibly with supplementary constraints) in the blocks world. Thus we first discuss "dedicated" methods for implicative and sv-constraints, and then methods for inferring combinations of the two types of constraints. We have formal proofs of correctness for all of our methods, but will have to restrict ourselves herein to some sketchy indications why the inferred constraints are correct.

## Implicative Constraints

We first consider implicative constraints of the form

$(\phi \Rightarrow \psi)\sigma_1 ... \sigma_k)$,

where $\phi$, $\psi$, and $\sigma_1,...,\sigma_k$ are simple *literals*, i.e., negated or unnegated atomic formulas whose arguments are constants or variables. Such constraints are to be interpreted as saying "In every state, for all values of the variables, if $\phi$ then $\psi$, *provided that* $\sigma_1$, ..., and $\sigma_k$. We assume that the predicate in $\phi$ is a fluent predicate, while that in $\psi$ may be fluent or static, and those in $\sigma_1,...,\sigma_k$ are static. $\sigma_1,...,\sigma_k$ are *supplementary conditions*; they could of course be conjoined to $\phi$, but we keep them separate because of the way they are determined algorithmically. Writing $V(\chi)$ for the variables occurring in a literal $\chi$, we also assume (in accord with the earlier remarks) that $V(\psi) \subseteq V(\phi)$ and $V(\sigma) \subseteq V(\phi)$ for all $\sigma \in \{\sigma_1 ... \sigma_k\}$.

We restrict ourselves here to STRIPS-like operators as used, for example, in POP (Weld 1994) or Graphplan (Blum & Furst 1995), allowing for a list of (possibly typed) parameters, a set of precondition literals and a set of effect literals. We define *persistent preconditions* as those precondition literals of an operator (including any type constraints on parameters) that do not resolve against any effect literals of the operator. Here resolution is understood as constrained by NEQ preconditions: a variable $x$ is considered unifiable with a constant or variable $\tau$ *unless* there is a precondition $\text{NEQ}(x,\tau)$.[1]

For example, if an operator $\text{STACKUP}(x,y)$ has preconditions

$\text{CLEAR}(x)$, $\text{CLEAR}(y)$, $\text{ON}(x,\text{TABLE})$, $\text{NEQ}(x,y)$, $\text{NEQ}(y,\text{TABLE})$

and effects

$\neg\text{ON}(x,\text{TABLE})$, $\text{ON}(x,y)$, $\neg\text{CLEAR}(y)$,

then $\text{CLEAR}(x)$ is persistent, since it doesn't resolve against the effect $\neg\text{CLEAR}(y)$, in view of the precondition $\text{NEQ}(x,y)$. $\text{NEQ}(x,y)$ and $\text{NEQ}(y,\text{TABLE})$ are also persistent, since no EQ condition appears among the effects of this (or indeed any) operator.

The number of syntactically legal implicative constraints can be very large.[2] However, at least in the relatively simple worlds that have so far served as test domains for domain-independent planners, the majority of useful implicative constraints seem to involve no more than two fluent literals, possibly along with some static literals (such as type contraints or EQ/NEQ-constraints on predicate arguments). Here fluent literals are literals whose predicates occur in operator effects, while static literals are ones whose predicates occur only in the initial conditions or in operator preconditions (including EQ/NEQ conditions, and type constraints on operator parameters). Even with an upper bound of two fluent literals, we cannot afford to consider all syntactically possible constraints, so we focus on those for which we have *a priori* evidence in the form of co-occurring literals in an operator.

The idea behind our basic algorithm for inferring implicative state constraints of the form $\phi \Rightarrow \psi$ (for the moment ignoring supplementary conditions) is the

[1] We assume here that there are no EQ-preconditions, so we needn't worry about preconditions $\text{NEQ}(x,\tau')$ where $\tau'$ is equated directly or indirectly to $\tau$.

[2] In particular, in a domain containing $m$ constants and $n_i$ predicates of arity $i$, $i \leq k$, there are more than $\Sigma_{i=0}^{k} 2n_i m^i$ literals (there are more than that because arguments may be variables as well as constants). Thus there are more than $N^l$ formulas with $l$ literals, where $N$ is the previous sum.

following. We hypothesize constraints by choosing $\phi$ to be an effect of an operator and choosing $\psi$ to be an effect or persistent precondition of that operator, where $V(\psi) \subseteq V(\phi)$. Restricting $\phi$ to operator effects ensures that it will be a fluent literal. We then attempt to verify the constraints.

To arrive at verification conditions, we observe that $\phi \Rightarrow \psi$ will hold in all reachable states if

1. It is true initially, for all substitutions of constants in the planning domain for the variables $V(\phi)$; and
2. Whenever some instance of $\phi$ becomes true as a result of applying an operator, the corresponding instance of $\psi$ will also be true in the resultant state; and
3. Whenever some instance of $\psi$ becomes false as a result of applying an operator, all corresponding instances of $\phi$ will also be false in the resultant state. (Since $\phi$ may have more variables than $\psi$, there may be many possible instances of $\phi$ corresponding to an instance of $\psi$.)

Concerning (1), suppose that we write the hypothetical implicative constraint as a clause $\lambda_1 \vee \lambda_2$. To verify this in the initial state, we must be able to verify at least one of the two literals for each way of substituting domain constants for the free variables. A substitution instance of a positive literal is verified if and only if it appears explicitly in the initial conditions or is of the form $\mathrm{EQ}(\kappa, \kappa)$, where $\kappa$ is a constant. (We count NEQ-literals as negative, and assume that $\neg$NEQ is transformed to EQ in the clausal form of a constraint.) A substitution instance $\neg\alpha$ of a negative literal (where $\alpha$ is not an EQ-atom) is verified if and only if $\alpha$ does *not* appear in the initial conditions; $\mathrm{NEQ}(\kappa, \kappa')$ (or $\neg\mathrm{EQ}(\kappa, \kappa')$) is verified if and only if $\kappa$ and $\kappa'$ are distinct constants. Note that we have used a closed world assumption (CWA) and a unique names assumption in the initial conditions.

We can easily translate these observations into an algorithm for verifying a hypothetical state constraint (in clause form) in the initial state. We iterate through substitutions for the free variables of the constraint, trying to verify the first literal, and if that fails, the second literal for each substitution. If for some substitution, the verification fails for both literals, the hypothesis is refuted and we can cut short the iteration. The generalization to the case where there are $k$ supplementary conditions is obvious: we will have a clause $\lambda_1 \vee ... \vee \lambda_{k+2}$ instead of a binary clause, and for each substitution we will try to verify one of the $k+2$ literals.

Concerning (2), suppose that an operator has a declared effect $\phi'$ which unifies with $\phi$ (subject to NEQ-preconditions), with unifier $u$. (Besides substituting constants for variables and operator parameters, the unification is allowed to substitute operator parameters for variables of $\phi$, never *vice versa*.) Then a sufficient (and easily confirmed) condition for (2) to hold is that $\psi_u$ is an effect or a persistent precondition of the operator, where $\psi_u$ is the result of applying substitution $u$ to $\psi$.[3]

Finally, (3) is trivially true if no operator has an effect matching $\overline{\psi}$ (complement of $\psi$). In addition, in cases where $V(\phi) = V(\psi)$, (3) can be verified just like (2) except that it applies to operators with a declared effect matching $\overline{\psi}$ with unifier $u$, and we look for persistent precondition or effect $\overline{\phi}_u$. If neither case applies, we need a different technique altogether (described later). For both (2) and the special cases of (3), the verification algorithm consists of scanning operator effects for effects that unify with $\phi$ or $\overline{\psi}$, and checking whether the corresponding persistent preconditions or effects $\psi_u$ or $\overline{\phi}_u$ are present.

When we hypothesize a state constraint $\phi \Rightarrow \psi$ based on an operator, the static preconditions of the operator (not involved in $\psi$) are good candidates as supplementary conditions. For instance, the law may only hold when the specified type constraints on the parameters hold, or a specified NEQ-precondition holds. Thus when hypothesizing a state constraint based on an operator we initially include the static preconditions of that operator as candidate supplementary conditions.

In testing a state constraint augmented with candidate supplementary conditions, we need to do some extra book-keeping if condition (2) or (3) is found to be violated. In particular, if condition (2) is violated, we keep track of (a) the name of the operator and name of the effect $\phi'$ for which we did not find a matching instance of $\psi$, and (b) the names of the candidate supplementary conditions which are *denied* in the preconditions of the operator, under substitution $u$. Note that any false supplementary condition can "rescue" the seemingly violated state constraint, since a conditional formula with a false condition is automatically true. If condition (3) is violated, we save analogous (a)-data and (b)-data.

After collecting all such (a)-data and (b)-data by going through all the operators, we try to select a minimal set of supplementary conditions sufficient to "rescue" the hypothetical state constraint relative to all the apparent violations that were encountered. The problem of finding a true minimal set turns out to be a variant of the (NP-hard) set-cover problem, so we do not insist on finding a true minimal set. Rather, we use a kind of greedy algorithm to find a small set of supplementary conditions sufficient to "excuse" all apparent violations. (The user can provide an upper bound on the permissible number of supplementary conditions.) If no such set of conditions is found, the hypothetical state constraint is discarded.

We have proved the computation of implicative constraints according to the above scheme to be

[3] Note that this sufficient condition is not necessary. It turns out that in order to obtain a necessary and sufficient condition we would have to generate the entire state space of the problem, to determine the exact set of possible parameter settings of each operator. This is in general impractical.

polynomial-time, in terms of the total number of preconditions and effects on the operators, and the number of atoms and constants in the initial conditions (assuming a fixed bound on predicate arity).

## Single-Valuedness Constraints

Next we describe a simple, "dedicated" method for finding sv-constraints. In the next section we will see how to infer implicative and sv-constraints in combination.

The idea of the dedicated method is to scan operators for effect literals that occur in pairs of the form $P(\tau_1,...,\tau_k)$, $\neg P(\tau_1,...,\tau_{i-1},\tau',\tau_{i+1},...,\tau_k)$ (with $\tau_i \neq \tau'$). For such pairs, we conjecture single-valuedness of $P$ in its $i$th argument.

We then check (a) whether positive $P$-occurrences in effects *always* occur paired in this way with negative $P$-occurrences, and in all cases we also have $P(\tau_1,...,\tau_{i-1},\tau',\tau_{i+1},...,\tau_k)$ among the preconditions; and (b) whether the single-valuedness property under consideration holds for the positive $P$-occurrences in the initial state; i.e., there are no pairs of instances of the form $P(\tau_1,...,\tau_k)$, $P(\tau_1,...,\tau_{i-1},\tau',\tau_{i+1},...,\tau_k)$, (with $\tau_i \neq \tau'$) in the initial state.

Note that condition (a) ensures that the "old" $P$-instance, with $\tau'$ instead of $\tau_i$ as $i$th argument, goes from true to false. We need to check the truth of the old $P$-instance in the preconditions, since otherwise that instance may have been false already. In that case it is not *made* false by the action, i.e., the action only *adds* a new $P$-instance, and this may destroy single-valuedness.

The suggested method of hypothesizing sv-constraints misses predicates that occur in the initial conditions but not in effects. However, we can easily identify such predicates, and see if they are single-valued initially (and hence throughout the state space).

So under these conditions we conclude that

$P(x_1,...,x_{i-1},*,x_{i+1},...,x_k)$.

Once again the techniques described are polynomial time-bounded, in the previous sense. Also it is possible to allow for supplementary conditions as in the case of implicative constraints, but in this case we have not pursued the matter.

## Inferring Combinations of Constraints

Blocks-world encodings often employ an operator MOVE$(x,y,z)$, meaning "move $x$ from $y$ to $z$". In such a setting, the constraint ON$(x,y) \Rightarrow \neg$CLEAR$(y)$ cannot be verified by the methods outlined earlier (even with a suitable supplementary condition on $y$). The reason is that there will be an effect CLEAR$(y)$ (the previous support block becomes clear), and condition (3) for the validity of implicative constraints then requires $\neg$ON$(x,y)$ to hold for *all* $x$, i.e., *nothing* is on $y$. But this universal is not explicitly asserted in the usual formulation of MOVE-effects; rather, the effects merely assert $\neg$ON$(x,y)$ for the *particular* $x$ that was moved.

We can overcome this difficulty by simultaneously inferring ON(*,$y$), i.e., at most one thing can be on $y$. For in that case, if just one particular $x$ is moved off $y$, $y$ will indeed become clear. This observation leads to a general method of dealing with cases that our previous verification conditions for (3) couldn't handle: if for the hypothesized implicative constraint $\phi \Rightarrow \psi$, $V(\psi) \subset V(\phi)$ and $\overline{\psi}$ matches some operator effect, then try to establish simultaneously that the predicate in $\phi$ is single-valued in its "extra" arguments, $V(\phi) - V(\psi)$. (This means that for each assignment of values to $V(\phi) \cup V(\psi)$, there is at most one satisfying assignment of values to $V(\phi) - V(\psi)$.)

We have formulated the following verification conditions for such simultaneous constraints, and proved them correct (we set aside supplementary conditions):

1. Suppose that an operator has a declared effect $\phi'$ that matches $\phi$ (subject to NEQ-preconditions), with unifier $u$. Then $\overline{\psi_u}$ is a precondition and, $\psi_u$ is an effect; and if there is another declared effect $\phi''$ that matches $\phi$, with unifier $u'$, then if $u$ and $u'$ make distinct substitutions for the "extra" variables of $\phi$, they also make distinct substitutions for the shared variables of $\phi$ and $\psi$.
2. Suppose that an operator has a declared effect $\overline{\psi'}$ that matches $\overline{\psi}$, with unifier $u$ (subject to NEQ-preconditions). Then for some substitution $v$ for the extra variables of $\phi$, $\phi_{uv}$ is a precondition and $\overline{\phi_{uv}}$ is an effect of the operator.

In addition, we have formulated, and proved correct, verification conditions for simultaneous implicative and sv-constraints, for implicative constraints $\phi \Rightarrow \psi$ where both $\phi$ and $\psi$ may contain variables not shared between them. An example of such a combination of constraints from the Logistics domain (referred to in our experiments below) is the following:

$((\mathrm{IN}(x,*y) \Rightarrow \neg\mathrm{AT}(x,*z))\ \mathrm{OBJECT}(x))$,

i.e., if an object $x$ is in (a vehicle) $y$ then $x$ is at no (external) location; and an object can be in at most one vehicle, and in at most one location. However, we have only implemented the first of the above methods of inferring combined constraints.

## Predicate Domain Constraints

Constraints on the predicate domains can be computed by first constructing a *forward action graph* which is a simplified planning graph in the sense of Graphplan (Blum & Furst 1995). The main difference between a forward action graph and a planning graph is that the latter contains mutual exclusion relations between action nodes and fact nodes, which are not present in the former. Since the number of such exclusion relations at any level of the graph can be quadratic in the number of the nodes at that level, it follows that the forward action graph of a given planning problem can be computed much more efficiently than the corresponding planning graph, in both time and space.

Our program for constructing the forward action graph of a given planning problem is based on a simple modification of IPP's C-code for building a planning graph (Koehler *et al.* 1997),[4] in which we avoid computing and propagating mutual exclusion relations. The depth of the action graph can be determined either by explicitly setting a parameter before running the program, or by letting the graph construction run until the graph "levels off" at some time step $i$ (the facts of level $i-1$ are the same as those of level $i$, and further extension of the graph cannot introduce any new fact (Blum & Furst 1995)).

The computation of the forward action graph using this simple method was considerably faster than the computation of the corresponding planning graph. For example, for the bw-large.c problem in the blocks world domain, which is examined in the next sections, it was about twenty times faster.

From the nodes of the fact levels of a forward action graph, we can extract information about the possible domains and codomains of the predicates appearing in the operator preconditions and effects. In fact, for any positive operator precondition or effect $\phi$, the only possible ground instances of $\phi$ that can hold at any reachable state in t time steps are a subset of the facts appearing as nodes of the action graph at level t. Therefore if a ground predication $\hat{\phi}$ does not appear at level t of the action graph, then under the CWA and the action closure assumption (Schubert 1994) (which are commonly used in most current planning systems) $\hat{\phi}$ is false at time t.

So, the *absence* of ground predications in the forward action graph can be used to formulate explicit constraints on the domains and codomains of the predicate arguments. Similarly, it is possible to use the action nodes of the forward action graph to derive constraints on the possible instantiations of the operators in the planning domain. However, in our preliminary experiments, which will be described in the next section, we have not yet tested the use of action variable constraints.

## Using State Constraints in SAT-based Planning

We have implemented our basic techniques for inferring implicative, single-valuedness and predicate domain constraints in the DISCOPLAN system.

Table I gives the implicative and sv-constraints computed by DISCOPLAN when applied to the blocks world problems used by Kautz and Selman (1996) to test their SATPLAN system. The domain formalization consists of two "move" operators, each of which contains two conditional effects, which we translated into four equivalent operators without conditional effects. Each object (including the table) is modeled as a block, and each object can be either "fixed" (the table) or "non-fixed" (an ordinary block). Moved objects must be non-fixed blocks, while source and destination objects can be either fixed or non-fixed.[5] The

```
(((IMPLIES (CLEAR ?X) (BLOCK ?X)))
 ((IMPLIES (CLEAR ?X) (NOT (FIXED ?X))))
 ((IMPLIES (NOT (BLOCK ?X)) (CLEAR ?X)))
 ((IMPLIES (ON ?*X ?Y) (NOT (CLEAR ?Y)))
           (NOT (FIXED ?Y)))
 ((IMPLIES (ON ?X ?Y) (BLOCK ?X)))
 ((IMPLIES (ON ?X ?Y) (BLOCK ?Y)))
 ((IMPLIES (ON ?X ?Y) (NOT (FIXED ?X))))
 ((IMPLIES (ON ?X ?Y) (NEQ ?X ?Y)))
 ((IMPLIES (NOT (ON ?X ?Y)) (BLOCK ?Y)))
 ((IMPLIES (NOT (ON ?X ?Y)) (BLOCK ?X))))
```

Table I: Implicative and sv-constraints discovered by DISCOPLAN for Kautz & Selman's formalization of the blocks world. Kautz & Selman's hand-coded constraints included the 4th, 7th and 8th implications, and the sv-constraint `(ON ?*X ?Y)`; they also posited that ON is non-commutative and that non-fixed blocks are ON just one thing, which DISCOPLAN currently does not infer.

CPU-time required to compute implicative and sv-constraints was 420 milliseconds on a SUN SPARC-station 4.

For the other domains that we tested the computation of these constraints required between 30 and 80 milliseconds, except for the Logistics domain. The state constraints inferable for this domain include implicative constraints $\phi \Rightarrow \psi$ where both $\phi$ and $\psi$ contain variables not shared between them. We did the computation of these constraints by hand, since the implementation of the corresponding method is still in progress. (We expect, however, that the computational time will not be significantly different from the cpu-times that we have given for the other domains).

Also the computation of the forward action graph was very efficient, and we believe it can be made even more efficient by omitting some data structures used by IPP that are not necessary for the construction of a forward action graph.[6]

In the rest of this section we describe the application of discovered constraints to SAT-based planning, and give preliminary empirical results illustrating their utility for this style of planning.

### Compiling discovered constraints into SATPLAN and MEDIC

In order to use the constraints discovered by DISCOPLAN in the context of SAT-based planning, we have extended two recent systems for compiling a planning problem specification into a set of propositional clauses: SATPLAN (Kautz & Selman 1996) and MEDIC (Ernst, Millstein, & Weld 1997).

While SATPLAN has its own planning language based on logical axiom schemas, MEDIC accepts

[4] IPP is of public domain and is available at `www.informatik.uni-freiburg.de/~koehler/ipp.html`.

[5] Note that the conjunction of the 1st and 3rd implication in Table I is consistent since every object of the domain, including the table, is predicated to be a block.

[6] The required cpu-time is dominated by the solution time in all 29 cases presented in the next section, except for Relsat applied to the easiest of the blocks worlds we considered. In this particular case Relsat solved the problem in 0.35 seconds, while the construction of the forward action graph required 1.91 seconds.

| Planning problems | Walksat | | | |
|---|---|---|---|---|
| | With constr. | Without constr. | Speedup factors | Failure reduction |
| bw-large.b | 17.7 | 95.5 | 5.4 | 0% |
| bw-large.c | 261 | 3,128-100% | > 59.2 | 100% |
| bw-large.d | 504 | 4,211-100% | > 41.7 | 100% |
| Hanoi4 | 1,238-100% | 1,200-100% | 0.97 | 0% |
| Ferry2-3 | 7.46 | 26.51-80% | > 3.5 | 80% |
| Ferry3-3 | 16.05 | 163.15-100% | > 50.8 | 100% |
| Ferry4-4 | 503.8-20% | 848.6-100% | > 1.6 | 80% |
| Monkey-test2 | 0.32 | 34.4 | 107 | 0% |

Table II: Average CPU-time (seconds) required by Walksat for solving encodings with and without the clauses derived from the discovered constraints (five runs for each problem). -n% indicates an n% failure rate in finding a solution. The last two columns indicate the improvements due to use of inferred constraints, in terms of cpu-time speedup and reduction in failure rates. Monkey-test3 is not included because it is unsolvable, and Walksat cannot detect this.

| Planning problems | Relsat | | | |
|---|---|---|---|---|
| | With constr. | Without constr. | Speedup factors | Failure reduction |
| bw-large.b | 0.35 | 7.85 | 22.42 | 0% |
| bw-large.c | 18.55 | 600-100% | > 33.3 | 100% |
| bw-large.d | 287.02 | 600-100% | > 2 | 100% |
| Hanoi4 | 79.69 | 600-100% | > 7.5 | 100% |
| Ferry2-3 | 0.17 | 1.65 | 9.7 | 0% |
| Ferry3-3 | 0.42 | 11.40 | 27.1 | 0% |
| ferry4-4 | 18.07 | 600-100% | > 33.2 | 100% |
| Monkey-test2 | 0.07 | 7.5 | 107.1 | 0% |
| Monkey-test3 | 0.29 | 600-100% | > 2,068 | 100% |
| Logistics.a | 3.63 | 4.35 | 1.19 | 0% |
| Logistics.b | 2.90 | 437.03-50% | > 150 | 50% |
| Logistics.c | 16.64 | 382.96-30% | > 23 | 30% |

Table III: Average CPU-time required by Relsat for solving encodings with and without the clauses derived from the discovered constraints (20 runs for each problem; each run had a search limit of 10 minutes). Times are in seconds. -n% indicates an n% failure rate (either no solution was found, though there was one, or the solver failed to establish unsolvability, though the problem was unsolvable). The last two columns indicate the improvements due to use of inferred constraints in terms of cpu-time speedup and failure rate reduction.

UCPOP-like specifications as input(Penberthy & Weld 1992), and offers the possibility of automatically compiling them using several representational settings (for details see (Ernst, Millstein, & Weld 1997)).[7] Our extended versions of SATPLAN and MEDIC expand the *un*simplified CNF files generated by SATPLAN and MEDIC respectively, by adding to them a set of clauses. These extra clauses are automatically computed by instantiating the discovered constraints. Each implicative and sv-constraint is instantiated to a set of ground clauses formed by literals appearing in the CNF file, and corresponding to initial conditions, action preconditions, action effects, and final goals (literals representing actions are not considered).

Predicate domain constraints can be extracted from a forward action graph in the following way. For each positive literal in the SAT encoding representing a *fluent*, we check whether the corresponding symbolic predication $P$ is a node of the action graph at the appropriate level. (Note that we automatically detect which predicates are static, and which are fluents during the computation of the implicative state constraints.) If this is not the case, then we generate the unit clause $\neg P$. The resulting set of unit clauses is then added to the SAT encoding together with the clauses derived from the implicative and sv-constraints.

The expanded SAT encodings can then be simplified by using a CNF simplifier such as SATPLAN's *Simplify* program or MEDIC's *Compact-cnf* program. In our tests we used Simplify version 2.1.

[7]In our tests we used the following settings, which in (Ernst, Millstein, & Weld 1997) were empirically shown to produce relatively small, easily solved encodings: regular action representation, explanatory frame axioms, parallel exclusion axioms, types elimination.

| Planning problems | Tableau | | |
|---|---|---|---|
| | With | Without | Speedup factors |
| bw-large.b | 5 | 65 | 13 |
| bw-large.c | 8 | > 26h | > 11,700 |
| bw-large.d | 5,260 | > 26h | > 18 |
| Hanoi4 | 26 | 13,371 | 514 |
| Ferry2-3 | 0.45 | 35 | 77 |
| Ferry3-3 | 0.74 | > 10h | > 48,640 |
| Ferry4-4 | 13 | > 8,400 | > 646 |
| Monkey-test2 | 0.53 | 0.57 | 1.07 |
| Monkey-test3 | 3 | > 10,800 | > 3,600 |

Table IV: CPU-time required by Tableau for solving encodings with and without the clauses derived from the discovered constraints. Times are in seconds, unless differently indicated ("h" means hours). > t means that the process was stopped after time t without reaching a solution, or without proving that the problem was unsolvable.

| Problems | Tableau | Walksat | Relsat | Vars | Clauses |
|---|---|---|---|---|---|
| bw-large.b | 2 | 23.8 | 0.53 | 1,087 | 13,772 |
| bw-large.c | 7 | 296 | 31.21 | 3,016 | 50,457 |
| bw-large.d | 4,503 | 2,047 | 395.99 | 6,325 | 131,973 |

Table V: CPU-time (seconds) and size of the encoding for the AT&T's blocks world problems as originally formulated, i.e., including hand-written domain axioms. The CPU-times for Walksat and Relsat are averages over 5 runs.

## Preliminary Empirical Results

Tables II, III, IV, V, VI give preliminary empirical results on the use of inferred constraints for improving SAT-based planning. For AT&T's blocks world (ATT-BW), we ran the following experiment.[8] First we removed from SATPLAN's specification of ATT-BW all the hand-written domain-specific axioms, and we tested the performance of several SAT solvers on encodings of the ATT-BW problems generated by SATPLAN for the revised specification. The revised encodings were very hard to solve not only for the stochastic solver Walksat (as already observed in (Ernst, Millstein, & Weld 1997)), but also for the systematic solver Tableau and for the more recent Relsat (Bayardo & Schrag 1997), which was tested using "relevance-bounded learning" of order 4. These results are given in the 3rd columns of Tables II, III and IV.

We then augmented the SAT encodings of these revised specifications with the clauses derived from the instantiation of our discovered constraints.[9] Finally, the resulting expanded encodings were first simplified and then solved using the three solvers mentioned above (for Tableau see the 2nd column of Table IV; for Walksat see the 2nd column of Table II; for Relsat see 2nd column of III). The results show that the use of the inferred constraints dramatically boosted speed and success in finding solutions regardless the kind of solver used. This is surprising since Kautz and Selman's experiments in (Kautz & Selman 1992) suggest that systematic SAT solvers based on the Davis-Putnam procedure such as Tableau could not be expected to benefit from addition of "axioms that explicitly rule out various impossible states" (Kautz & Selman 1992). The performance of Tableau for these encodings was very close to its performance for the original formalization of ATT-BW including hand-written domain-specific axioms (compare the 2nd columns of Table IV and V). The performance of Walksat and Relsat for these encodings was even better than the performance for the original ones (compare the 2nd columns of Tables II and III with the 3rd and the 4th of Table V respectively).

[8]All our tests were conducted on a SUN Ultra 1 with 128 Mbytes of RAM. The tests for Walksat were conducted with the following "cutoff" and "noise" settings. For the ATT-BW problems we used the settings suggested by Kautz and Selman; while for the other problems we used noise 30–100 and cutoff 2,000,000 for Hanoi4 and Ferry4-4, cutoff 500,000 for Ferry3-3 and cutoff 100,000 for Ferry2-3 (i.e., we used smaller cutoffs for easier problems).

[9]The constraints were inferred using a set of four STRIPS operators that we derived from SATPLAN's axiomatic formalization of the blocks world.

| DISCOPLAN constr. | CPU-time | Vars | Clauses |
|---|---|---|---|
| Implicative | 7 | 3,016 | 20,717 |
| All the constraints | 8 | 3,016 | 36,949 |
| Implicative and SV | 9 | 3,016 | 33,704 |
| Predicate Domain | 1,302 | 3,016 | 18,603 |
| Single-valuedness (SV) | $> 8{,}000$ | 3,526 | 38,340 |
| No constraints | $> 93{,}600$ | 3,526 | 21,957 |

Table VI: Average CPU-time (seconds) required by Tableau and size of the encodings for bw-large.c using each kind of discovered constraints.

In order to test the individual impact of each of the discovered constraints, we separately tested each of them on the ATT-BW problem bw-large.c (involving 15 blocks on a table). For each kind of constraint we generated a different SAT encoding formed by the unsimplified set of the clauses generated by SATPLAN (excluding those corresponding to all the hand-written domain specific axioms) and by the clauses derived from the discovered constraints of the kind under consideration. Similarly we tested the impact of the implicative constraints combined with single-valuedness. The solution times required by Tableau for solving these SAT instances are given in Table VI. They show that in ATT-BW the most useful discovered constraints are the implicative constraints. Interestingly, in this domain sv-constraints do not help when used alone, or in combination with implicative constraints. On the other hand, predicate domain constraints do help to guide the solver towards a solution, though not as effectively as implicative constraints.

It addition to the ATT-BW problems we tested the use of our techniques in four other well-known domains taken from the test suites of UCPOP, MEDIC or SATPLAN: Tower of Hanoi, Ferry, Monkey and ATT-logistics. The SAT encodings for these domains where automatically obtained using MEDIC. Hanoi4 is the Tower of Hanoi problem with four disks. Ferry-2-3 is a Ferry problem with two "autos" to be transported and three locations (analogously for Ferry-3-3 and Ferry-4-4). Monkey-test2 and Monkey-test3 are problems in the Monkey domain; the first is solvable, while the second is unsolvable. Finally, Logistics.a, Logistics.b and Logistics.c are the problems in the ATT-logistics domain that were used by Kautz and Selman to test their SATPLAN system, and which, like the ATT-BW problems, are very hard to solve for planners based on planning graphs (Kautz & Selman 1996; Koehler *et al.* 1997).

The results for these problems are given in Tables III and IV. They show that inferred state constraints can be extremely useful in these domains as well. In the tests for the Ferry and the Monkey problems we used the sv-constraints that are discovered both in combination with implicative laws, and by using the dedicated method that we discussed in the previous sections. The constraints derivable using the dedicated method are {(at-ferry *)}, for the Ferry domain, and {(AT monkey *)} for the Monkey domain. They represent the intuitive constraint that in any legal state the ferry and the monkey can be at most one location. Without these sv-constraints the performance of Tableau was significantly worse. For example, Ferry-3-3 was solved in 119 seconds, instead of 0.74. In contrast with the blocks world, this is an example of a domain where sv-constraints are important. The same is true for monkey-test3, in which we found that the sv-constraint for the location of the monkey is crucial in enabling Tableau to prove that the problem has no solution.

Concerning the ATT-Logistics problems, the use of state constraints considerably improved the performance of Relsat. However, for Tableau and Walksat these problems were hard to solve even with the addition of state constraints.

Finally, the fourth and fifth columns of Tables II and III give the speedup factors and the improvement in failure rate that we obtained for all the problems we tested. In almost all cases we obtained either a drastic speedup or a considerable reduction in failure rate, regardless the CNF solver used.

## Conclusions and Further Work

We have provided provably correct, polynomial-time methods for inferring a variety of state constraints in STRIPS-like planning domains. Our preliminary experiments show that automatically inferred constraints can be very effective in speeding up SAT-based planners.

Current and future work includes

- Generalization of our implemented methods to allow for UCPOP-like conditional effects; much of our code already allows for this, but the selection of supplementary constraints becomes more complicated;
- Implementation of additional methods in DISCOPLAN; some were already mentioned, and some not, such as exclusion and subsumption relations among static predicates, non-commutativity laws, and strict sv-laws;
- Further experiments to test the usefulness of automatically inferred constraints; and
- "bootstrapping" the discovery of constraints by feeding confirmed constraints back into the hypothesize-and-test process.

We also expect that inferred constraints will prove useful in other types of planners; for instance, they could be used for "unachievability pruning" in deductive or regression planners (Genesereth & Nilsson 1987). Past experience in AI attests that any search process can benefit from the imposition of local constraints on the search space.

## Acknowledgments

This research was supported in part by NATO grant CRG951285, by CNR project SCI*SIA (first author), and by ARPA/SSTO grant F30602-95-1-0025 (second author). We thank the anonymous referees for their helpful comments.

## References

Bayardo Jr., R., and Schrag, R.C. 1997. Using CSP look-back techniques to solve real-world SAT instances. In *Proc. of the 14th Nat. Conf. of the American Assoc. for Artificial Intelligence (AAAI-97)*, 203–208. Morgan Kaufmann.

Blum, A., and Furst, M. 1995. Fast planning through planning graph analysis. In *Proc. of the 14th Int. Joint Conf. on Artificial Intelligence (IJCAI-95)*, 1636–1642. Morgan Kaufmann.

Crawford, J., and Auton, L. 1993. Experimental results on the cross-over point in satisfiability problems. In *Proc. of the 11th Nat. Conf. of the American Assoc. for Artificial Intelligence (AAAI-93)*, 21–27.

Ernst, M.; Millstein, T.; and Weld, D. 1997. Automatic SAT-compilation of planning problems. In *Proc. of the 15th Int. Joint Conf. on Artificial Intelligence (IJCAI-97)*, 1169–1176. Morgan Kaufmann.

Genesereth, M.; Nilsson, N. 1987. *Logical Foundations of Artificial Intelligence.* Morgan Kaufmann.

Gerevini, A.; Schubert, L.K. 1996. Accelerating partial-order planners: Some techniques for effective search control and pruning, *J. of Artificial Intelligence Research (JAIR) 5*, 95–137.

Kautz, H., and Selman, B. 1992. Planning as satisfiability. In *Proc. of the 10th European Conf. on Artif. Intell. (ECAI-92)*, 359–363. John Wiley and Sons.

Kautz, H., and Selman, B. 1996. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proc. of the 13th Nat. Conf. of the American Assoc. for Artificial Intelligence (AAAI-96).*

Koehler, J.; Nebel, B.; J., H.; and Dimopoulos, Y. 1997. Extending planning graphs to an ADL subset. In *Proc. of the 4th European Conf. on Planning (ECP'97).* Springer Verlag.

Penberthy, J., and Weld, D. 1992. UCPOP: A sound, complete, partial order planner for ADL. In Nebel, B.; Rich, C.; and Swartout, W., eds., *Proc. of the 3rd Int. Conf. on Princ. of Knowledge Representation and Reasoning (KR'92)*, 103–114. Morgan Kaufmann.

Schubert, L. 1994. Explanation closure, action closure, and the Sandewall test suite for reasoning about change. *J. of Logic and Computation* 4(5):679–700.

Selman, B.; Kautz, H.; and Cohen, B. 1994. Noise strategies for local search. In *Proc. of the 12th Nat. Conf. of the American Assoc. for Artificial Intelligence (AAAI-94)*, 337–343. Morgan Kaufmann.

Weld, D. An introduction to least commitment planning. *AI Magazine*, 15(4), 27–62.