Discovering State Constraints in DISCOPLAN: Some New Results

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

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

Abstract

DISCOPLAN is an implemented set of efficient preplanning algorithms intended to enable faster domain-independent planning. It includes algorithms for discovering state constraints (invariants) that have been shown to be very useful, for example, for speeding up SAT-based planning. DISCOPLAN originally discovered only certain types of implicative constraints involving up to two fluent literals and any number of static literals, where one of the fluent literals contains all of the variables occurring in the other literals; only planning domains with STRIPS-like operators were handled. We have now extended DISCOPLAN in several directions. We describe new techniques that handle operators with conditional effects, and enable discovery of several new types of constraints. Moreover, discovered constraints can be fed back into the discovery process to obtain additional constraints. Finally, we outline unimplemented (but provably correct) methods for discovering additional types of constraints, including constraints involving arbitrarily many fluent literals.

Introduction

The automated inference of state constraints (invariants) based on a given set of state-transforming operators and initial conditions has emerged as a significant new development in the effort to design effective domain-independent planners (Kelleher & Cohn 1992; Rintanen 1998; Gerevini & Schubert 1996; 1998; Fox & Long 1998). It has been shown that such state constraints, derived in a pre-planning phase, can be used to greatly reduce the planning search space and hence planning time (Gerevini & Schubert 1996; 1998; Rintanen 1998; Kautz & Selman 1998; Fox & Long 1998; 2000; Refanidis & Vlahavas 2000), as well as to aid specification and debugging of planning domains.

In (Gerevini & Schubert 1998) we proposed a collection of techniques for extracting state constraints from a set of operators and an initial state, and we implemented a subset of these techniques in the DISCOPLAN (DIScovering COnstraints for PLANning) package. On the assumption that operators are given in STRIPS-like form, we implemented the derivation of (a) fluent predicate domains (sets of k-tuples that include all possible argument tuples for which a k-ary predicate may hold; these were obtained by a Graphplan-like method); (b) implicative constraints of form

`((IMPLIES` ϕ ψ`)` $\sigma_1 ... \sigma_n$`)`,

where $\phi, \psi, \sigma_1, ..., \sigma_n$ $(n \geq 0)$ are literals, the σ_i are static (non-fluent) supplementary conditions such as type and inequality constraints, and the antecedent literal ϕ contains all variables occurring elsewhere; an example is

`((IMPLIES (AT ?X ?Y) (AIRPORT ?Y)) (AIRPLANE ?X))`

(from the "`att-logistics`" world, stating that for all `?X`, `?Y`, if `?X` is at `?Y` then `?Y` is an airport, provided that `?X` is an airplane); and (c) simultaneous implicative and single-valuedness (sv-) constraints such as the blocks-world constraint

`((IMPLIES (ON ?*X ?Y) (NOT (CLEAR ?Y))) (NEQ ?Y T)`,

where "starred" variables indicate that there can be at most one value of the argument in the starred position corresponding to any given values for the remaining arguments of the predicate in question, and `T` stands for the table. In addition, we outlined (but did not implement) algorithms for deriving certain sv-constraints independently of implicative constraints, and for deriving simultaneous implicative and sv-constraints more general than those in (c), allowing both the antecedent and the consequent to contain variables not contained in the other.

Here we describe some major extensions of DISCOPLAN, in both practical and theoretical directions. An across-the-board extension is the allowance for *conditional* effects in operators specifying a planning domain. This is likely to be of considerable practical interest in domain-independent planning, since formalisms permitting operators with conditional effects (e.g., UCPOP and PDDL) facilitate compact encoding of complex operators that would otherwise require unnatural and potentially very large expansions as multiple unconditional operators.

The algorithms and implementation for (b) and (c) above have been generalized accordingly, and the newly implemented and newly proposed techniques similarly allow for conditional effects. These extensions are described in the second section. In the third section, we describe how DISCOPLAN is able to infer additional constraints of the type just mentioned by "expanding" operators so as to include preconditions and effects implied by constraints discovered earlier, and then re-running the discovery algorithms. In the forth section, we describe some further extensions to our methods (mostly unimplemented, but provably correct); this includes methods for inferring strict single-valuedness (and n-valuedness) constraints, and constraints involving arbitrarily many fluent literals. We provide sample results in the fifth section, and then summarize our contribution and its relation to other approaches, and our further plans.

Newly Inferred Constraints

We begin by outlining the generalizations needed to deal with conditional effects in deriving implicative and simultaneous implicative and sv-constraints (a slightly more detailed description is given in (Gerevini & Schubert 2000)). Then we devote a series of subsections to the new capabilities of DISCOPLAN.

Adapting the Hypothesize-and-Test Paradigm to Conditional Effects

We assume operators similar to those handled by UCPOP, except that we do not allow for universal quantification. After (automatic) standardization, each operator consists of a name, a set of parameters, and a set of *when*-clauses. Each *when*-clause contains a set of precondition literals and a set of effect literals, any of which may have constant or parametric arguments and may be positive or negated. The first *when*-clause, called the *primary when*-clause, contains the preconditions that must be verifiable whenever the operator is applied to a state, and the effects whose truth is assured in the resulting state. Each of the remaining, *secondary when*-clauses (if any) specifies additional preconditions and effects, where satisfaction of those preconditions along with the primary ones assures the truth of those effects in the resulting state.

We use several top-level programs to infer different types of constraints from operator structure, but most of them adhere to a hypothesize-and-test paradigm with the following structure. (An exception is the program for finding antisymmetry constraints, in which the first of the following steps is based on single literals rather than pairs). Note that Γ can be logically complex, for instance consisting of both an implicative hypothesis and one or two sv-hypotheses. This is crucial since some hypotheses cannot be verified in isolation, but only by simultaneous induction with other hypotheses.

1. Hypothesize a constraint Γ based on co-occurrences of literals in a *when*-clause w of an operator and in the corresponding primary *when*-clause w_1 (if different). For example, effects ϕ and ψ might lead to an implicative hypothesis (`IMPLIES` ϕ ψ), and possibly sv-hypotheses about the predicates involved.
2. Add a set of candidate supplementary conditions $\{\sigma_1, ..., \sigma_n\}$, consisting of the static preconditions of w and w_1 and if $w \neq w_1$, the negations of static preconditions of other *when*-clauses (except ones that unify with static preconditions of w or w_1 or their negations).
3. Test hypothesis Γ relative to each *when*-clause of each operator, using the relevant *verification conditions*; for each apparent violation of Γ find the corresponding possible "excuses" for the violation. An excuse is a set of provisos $\{\sigma'_1, ..., \sigma'_m\}$, chosen from the candidate supplementary conditions, that weaken the hypothesis sufficiently to maintain is truth. If a violation has no excuses, abandon the hypothesis Γ, otherwise record the set of possible excuses of the violation on a global list.
4. Find all minimal subsets (up to a given size)[1] of $\{\sigma_1, ..., \sigma_n\}$ that "cover" all apparent violations of Γ; a subset of $\{\sigma_1, ..., \sigma_n\}$ covers an apparent violation of Γ if it contains all elements of at least one "excuse" for that violation;
5. Check hypothesis (Γ $\sigma'_1 ..., \sigma'_m$) (i.e., the original hypothesis together with added provisos) for each of the minimal subsets $\{\sigma'_1, ..., \sigma'_m\}$ of $\{\sigma_1, ..., \sigma_n\}$ found in the previous step for truth in the initial conditions of the problem being solved; return the variant hypotheses that pass this test as the verified hypotheses.

Verification Conditions and "Excuses"

Verification conditions are conditions on the precondition-effect structure of operators that are needed to support an inductive proof that the operators maintain a given type of state constraint. (We have such inductive proofs for the constraints found by DISCOPLAN, but space does not allow their inclusion.) The most complex aspect of the above sequence of steps is the collection of possible "excuses" (sets of candidate supplementary conditions) in step 3, when verification conditions for Γ are violated. The details depend on the verification conditions (as determined by the form of Γ), and different verification conditions call for different "excuses" when violated. However, there are commonalities across the various types of hypotheses that are exploited in our code. In particular, there are essentially 4 types of verification conditions: (i) ones that preclude the occurrence of multiple effects of the same type (e.g., in testing `(ON ?*X ?Y)`, we want to guard against multiple effects such as `(ON ?U ?W)`, `(ON ?V ?W)`); (ii) ones that require the co-occurrence of a certain type of effect with another (e.g, in testing an implicative constraint, whenever an effect instantiates the antecedent, another effect should instantiate the consequent; and similarly for the contrapositive of the implication); (iii) ones that require a *change* (a certain type of precondition and a related effect) whenever a given type of effect is present (e.g., this is needed whenever Γ involves sv-constraints); and (iv) ones that preclude the co-occurrence of certain effects with certain other effects (e.g., this is needed in testing exclusion hyotheses, stating that the truth of one predication implies the falsity of another). Our programs for testing hypotheses and collecting "excuses" are organized around 4 subroutines corresponding to these 4 types of verification condition.[2]

The "excuses" themselves are essentially of two types. One type of "excuse" ensures that a particular *when*-clause of an operator is rendered irrelevant to a hypothesis. In this case the excuse is a singleton $\{\sigma\}$ (chosen from the candidate supplementary conditions) whose falsity is entailed by the preconditions of that *when*-clause (or by the preconditions of the corresponding primary *when*-clause, if differ-

[1] For the domains we have tested a size limit of 2 suffices; allowance for up to 5 supplementary conditions yielded no new constraints.

[2] Actually, we have recently added a fifth routine testing for the presence of a "compensating change", which allows us to make stronger use of the induction assumption in testing implicative constraints. We briefly discuss this later.

ent). This type of excuse is considered whenever a *when*-clause generates an effect that should not co-occur with another given effect, or whenever it generates an effect whose co-occurrence requirements are not (provably) met. The second type of "excuse" ensures that the effects of a particular secondary *when*-clause are realized. In this case the "excuse" may contain multiple elements of $\{\sigma_1, ..., \sigma_n\}$, which together entail the preconditions of that *when*-clause. This type of "excuse" is considered whenever a required co-occurring effect is not guaranteed in the *when*-clause under consideration, but can be guaranteed *via* the effects of another *when*-clause, if the preconditions of that other *when*-clause are true. The search for "excuses" for apparent violations of hypothesized constraints has recently been strengthened by the use of subtype/supertype and exclusion relations among static monadic predicates. (These are found in the manner explained in the subsection that follows.)

We now provide some specifics of the discovery process for most types of constraints in the new version of DISCOPLAN, with particular reference to verification conditions (where relevant). We omit discussion of simple implicative constraints (without sv-constraints).

Static Constraints

Static constraints are state invariants involving *type-predicates*, where a type-predicate is a static monadic predicate that occurs positively in the initial state. In general, the result of this analysis gives the following information about types:

- a list of type-predicates, each of which is associated with the set of objects of the domain satisfying the predicate;
- a list of *universal types* (i.e., predicates that are satisfied by every object in the domain), and a list of *empty types* (i.e., predicates that are satisfied by no object);
- a list of supertype/subtype and incompatible relationships between type-predicates.

Type information is computed in polynomial time by processing the initial state in the following way (the algorithm assumes that for each predicate P in the domain it is known whether P is static – this information is computed during the initial standardization of the operators). First we compute the set Θ of the constants appearing in the specification of the initial state, and we associate with each type-predicate the set of the constants appearing in the positive instances of P.[3] This set, indicated with $|P|$, is the *extension* of P. If we have that $|P| = \Theta$, then P is an universal type, while if $|P| = \emptyset$, then P is an empty type; if for some other type-predicate Q, we have that $|P| \subseteq |Q|$, then P is a subtype of Q and Q is a supertype of P, i.e., each object of type P must be of type Q; if $|P| \cap |Q| = \emptyset$, then P and Q are incompatible types, i.e., no object in Θ can be both of type P and of type Q. For example, suppose that the list of static predicates appearing in the initial state is:

```
((P a) (P b) (Q b) (R a) (S a) (S b)
 (S c) (T a b) (T b c)).
```

The following information is computed by DISCOPLAN:

- $\Theta = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$; Type-predicates: `(P Q R S)`
- $|\mathtt{P}| = \{\mathtt{a}, \mathtt{b}\}$, $|\mathtt{Q}| = \{\mathtt{b}\}$, $|\mathtt{R}| = \{\mathtt{a}\}$, $|\mathtt{S}| = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$
- Universal types: `(S ?X)`; empty types: `nil`
- Super/sub-type relationships:
```
((IMPLIES (Q ?X) (P ?X)) (IMPLIES (R ?X) (P ?X))
 (IMPLIES (P ?X) (S ?X)) (IMPLIES (Q ?X) (S ?X))
 (IMPLIES (R ?X) (S ?X)))
```
- Incompatible types:
```
((IMPLIES (Q ?X) (NOT (R ?X)))).
```

[3] Here we assume that all the objects (constant symbols) of the domain appear in the initial state as terms of some positive literal.

Dedicated Inference of SV-Constraints

An example of an sv-constraint that can be inferred in isolation is the blocks-world constraint `((ON ?X ?*Y))`, i.e., any object can be `ON` at most one other object; or, in unabbreviated FOL,

$\forall x, y, z.(ON(x, y) \wedge ON(x, z)) \Rightarrow y = z.$

As an example involving a supplementary condition (from the `logistics-att` world), we have

```
((AT ?X ?*Y) (AIRPLANE ?X)),
```

i.e., an airplane can be `AT` no more than one place. Our "dedicated" method of finding sv-constraints of this type starts by forming hypotheses based on the occurrence within a *when*-clause and the corresponding primary *when*-clause of an effect $(P\ t_1...t_n)$ together with a "compensating change", i.e., a P-precondition and corresponding $\neg P$-effect that appears to maintains single-valuedness. The verification conditions ensure (a) that there are no multiple effects that could violate single-valuedness, and (b) that any P-effect is indeed accompanied by a "compensating" change. Violations lead to collection of "excuses" of the sort indicated earlier (if possible), and these provide the basis for deriving minimal sets of supplementary conditions. (a) and (b) suffice for an inductive proof that if a constraint such as `((P ?X ?*Y) (Q ?X))` holds in the initial state, it holds in all reachable states. (Additional starred and unstarred arguments and multiple supplementary conditions are easily dealt with.)

Implicative Constraints + SV-Constraints: The Case of Subsumed Variables

In the introduction, we mentioned the blocks-world constraint

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

as an example of a combined implicative and sv-constraint obtainable by the previous version of DISCOPLAN. In general, the implicative constraints we are considering here have as their antecedent a positive literal that contains at least one "starred" variable not occurring in the consequent, and zero or more "unstarred" variables occurring in the consequent. The stars indicate that for all values of the unstarred variables, the antecedent holds for at most one tuple of values of the starred variables.

The discovery of such constraints in the previous version of DISCOPLAN was limited to domains where operator effects are unconditional. The new algorithm for operators with conditional effects proceeds much as before, though of course with complications due to the fact that multiple *when*-clauses may contribute to the effects of an operator. Potential antecedent-consequent pairs are hypothesized based on co-occurrence of a positive effect literal ϕ with another effect or persistent precondition ψ whose variables are a proper subset of those of ϕ. The complement of the signed predicate of ψ must occur as an effect of some operator (otherwise simultaneous inference of an sv-constraint would be unnecessary), and ϕ and ψ must belong to the pooled effects and persistent preconditions of some *when*-clause w and the corresponding primary *when*-clause w_1. For the special case of an antecedent `(P ?X ?*Y)` and consequent `(Q ?X)` the verification conditions are the following (these are easily generalized to allow for multiple shared and starred variables):

(a) There must not be multiple effects matching `(P ?X ?*Y)` that could directly violate the sv-constraint. The details are as in condition (a) for "dedicated" sv-testing.

(b) If w contains an effect (P x y) for some x and y, there must be a precondition (NOT (Q x')) and an effect (Q x'') in w or the corresponding primary *when*-clause w_1, where $x'' = x' = x$ by symbol identity or EQ-preconditions. (We have recently weakened this condition in a way that makes stronger use of the induction assumption, but leave out details for simplicity.)

(c) If w contains an effect (NOT (Q x)) for some x, there must be a precondition (P x' y) and an effect (NOT (P x'' y')) in w or w_1, where $x'' = x' = x$ and $y = y'$ by symbol identity or EQ-preconditions.

Implicative Constraints + SV-Constraints: The Case of Non-Subsumed Variables

In the implicative constraints considered in the preceding subsection, the antecedent variables were required to subsume the consequent variables. Here we assume instead that both antecedent and consequent contain variables not contained in the other. All such variables are "starred", while the shared variables are unstarred. An example is the following constraint from the `Logistics` world:
`((IMPLIES (AT ?X ?*Y) (NOT (IN ?X ?*Z))) (OBJ ?X))`.
This is an *exclusive* state constraint, i.e., it states that no object can simultaneously be AT something and IN something (and in addition an object can be AT no more that one thing, an IN no more than one thing). In (Gerevini & Schubert 1998) we formulated provably correct criteria for deriving such constraints, but once again we did not generalize to operators with conditional effects. The generalized method proceeds much as in the case of implicative constraints with subsumed variables (previous subsection), and we need only point out the main differences. First, hypothesis formation relies on literal co-occurrences as before, except that both the antecedent and consequent are based on effects (with no consideration of persistent preconditions) and the restrictions relevant to exclusive state constraints are placed on signs and variables. There are five verification conditions; for the (easily generalized) special case of an exclusive state constraint with antecedent `(P ?X ?*Y)` and consequent `(NOT (Q ?X ?*Z))`, they run as follows. The first two conditions guard against multiple effects matching `(P ?X ?*Y)` or `(Q ?X ?*Z)`, as in the previous condition (a). The third condition states

(c) If w contains an effect (P x y) for some x, y, there must be a precondition (Q x' z) and an effect (NOT (Q x'' z')) in w or w_1, where $x'' = x' = x$ and $z = z'$ by symbol identity or EQ-preconditions.

The fourth condition is completely analogous to the third, with P and Q and y and z interchanged. The fifth and final condition guards against co-occurrence of an effect (P x y) in one *when*-clause with an effect (Q x' z) in the same or in another *when*-clause, where $x = x'$ by symbol identity or EQ-preconditions. This condition is most easily understood by thinking of an exclusive state constraint as a disjunction of two negative literals, and recognizing that if both literals become false for the same shared argument and some values of the non-shared arguments, then the disjunction cannot remain true for all values of the three variables.

Antisymmetry Constraints

Antisymmetry constraints (as-constraints) are particular implicative constraints of the form

((IMPLIES (P t_1 t_2) (NOT (P t_2 t_1))) σ_1 $\sigma_2 ... \sigma_n$),

where t_1 and t_2 can be constants or universally quantified variables, and $\sigma_1, ..., \sigma_n$ are supplementary conditions whose variables are a subset of $\{t_1, t_2\}$. An example of an antisymmetry constraint in the blocks-world is

`((IMPLIES (ON ?X ?Y) (NOT (ON ?Y ?X))))`,

i.e., if one object is on another, then the second is not on the first. Like the previous methods, the method for discovering antisymmetries uses the hypothesize-and-test paradigm described above. In particular, if (P t_1 t_2) is an effect of a *when*-clause in an operator op, then we hypothesize ((IMPLIES (P t_1 t_2) (NOT (P t_2 t_1)))). The hypothesis is then augmented with candidate supplementary conditions in the same manner as for implicative constraints, and tested against the operators to yield variants augmented with minimal sets of supplementary conditions, and finally these variants are tested in the initial state. The verification conditions for an antisymmetry hypothesis Γ, enabling an inductive proof that Γ holds in all reachable states, are the following:

For each when-clause w of each operator o, if w has an effect matching (P t_1 t_2), then assuming that (a) Γ is true in any state s where o is applied and the preconditions of w hold, and (b) Γ becomes false in the state s' resulting by applying o to s, leads to a contradiction (because s or s' would have to be inconsistent).

In order to test this condition, for each *when*-clause w of o, if w has an effect matching (P t_1 t_2) with unifier u, we add $[(P\ t_2\ t_1)]_u$ to the effects of w. Then we test each expanded *when*-clause w_e to see whether the set Ω formed by the effects of w_e, together with the persistent preconditions and effects of the primary *when*-clause w_1 of o (if $w_e \neq w_1$), is *in*consistent. If this is the case, then Γ is confirmed for w. If this is not the case, then we collect sets of supplementary conditions that can excuse the violation of the verification condition.

To strengthen this method, when we test the verification condition against an operator, we use an "expanded" version of the operator, obtained by augmenting the preconditions and effects of each *when*-clause using implicative constraints discovered earlier. (This process is described in the next section). The resulting Ω-sets are inconsistent if Ω contains a pair of contradictory non-static conditions, contradictory EQ/NEQ-conditions, or a pair of static conditions that violate static constraints.

XOR-constraints

XOR-constraints are state-constraints of the form

((XOR ϕ ψ) σ_1 $\sigma_2 ... \sigma_n$),

where ϕ and ψ are positive fluent literals, such that non-shared variables are existentially quantified, while shared variables are universally quantified, and where the variables in σ_1, σ_2, ..., σ_n can only be variables shared by ϕ and ψ. An example of an XOR-constraint in the logistics domain is

`((XOR (AT ?X ?Y) (IN ?X ?Z)) (OBJECT ?X)),`

stating that in any reachable state, any object is either `at` some place or `in` something. The method that we have developed for inferring this type of constraint is based on combining two types of constraints entailing exclusive disjunctions. The first type of constraint consists of those implicative constraints inferred by our previously described methods where ϕ and ψ are fluents, ϕ is positive and ψ is negative. The second type of constraint corresponds to binary "state membership invariants" (Fox & Long 1998). These are binary disjunctions, possibly augmented with supplementary conditions, of the following form ((OR ϕ ψ) σ_1 $\sigma_2 ... \sigma_k$), where the non-shared variables of ϕ and ψ are existentially quantified, and the remaining variables are universally quantified. Our method for inferring state membership invariants is a variant of the methods for inferring implicative constraints described in previous sections (for lack of space we omit a detailed description). One of the main differences lies in the way hypotheses are verified in the initial state, and in the way supplementary conditions capable of rescuing the law are collected. When we check a hypothetical membership invariant against the initial state, we consider additional type predicates (not appearing in the operator from which the hypothesis was derived) which restrict the domains of universally quantified variables in ϕ and ψ. For example, in UCPOP's formalization of the *Ferry* domain DISCOPLAN can infer the following membership invariant

`((OR (AT ?X ?Y) (ON ?X FERRY)) (AUTO ?X)),`

where `(AUTO ?X)` is a supplementary condition that is required for verifying the law in the initial state, and which does not belong to any operator that suggested the hypothesis. By combining this state membership constraint with the exclusive constraint

`((IMPLIES (AT ?X ?Y) (NOT (ON ?X FERRY)))),`

DISCOPLAN infers the XOR-constraint

`((XOR (AT ?X ?Y) (ON ?X FERRY)) (AUTO ?X)).`

Using "Expanded" Operators

As mentioned above DISCOPLAN discovers as-constraints using expanded operators. Moreover, as pointed out in (Gerevini & Schubert 1998), the general hypothesize-and-test process can be enhanced by feeding confirmed constraints back into the process. A straightforward way to do this is to expand each operator by adding extra preconditions and effects that are implied by constraints discovered earlier. In particular, the current version of DISCOPLAN expands the given operators using implicative constraints with subsumed variables.[4]

[4]Note that in the current implementation type constraints are not used for expanding the operators. Reasoning about type constraints is incorporated into the various discovery routines.

By re-running all the discovery algorithms using the expanded operators, further constraints can be derived. This process of inferring state-constraints and expanding the operators using discovered constraints can be repeated until no more new constraints are derived.

An operator o is expanded by using each implicative constraint ((IMPLIES ϕ ψ) $\sigma_1, \sigma_2, ..., \sigma_n$) with subsumed variables in the following way. For each *when*-clause w of o, if a precondition or effect χ of w matches ϕ with unifier u, and $\sigma_1 ... \sigma_n$ are satisfied under u, then we augment w with $[\psi]_u$. Specifically, we add $[\psi]_u$ to the preconditions of w, if χ is a precondition of w or ψ is a static condition; while we add $[\psi]_u$ to the effects of w if χ is an effect of w and ϕ is non-static. Writing w_1 for the primary *when*-clause of o, if $w = w_1$, then the check for the validity of $\sigma_1 ... \sigma_n$ is done against the preconditions of w; otherwise (w is a secondary *when*-clause) the supplementary conditions are checked against the preconditions of w extended with the preconditions of w_1. Also, note that if $w = w_1$ and some supplementary preconditions are not satisfied by the preconditions of w_1, then it is still possible that such conditions are satisfied by the preconditions of a secondary *when*-clause w_2. If this is the case, then $[\psi]_u$ is added to the preconditions or effects of w_2 (depending on the conditions indicated above).

Finally, if a *when*-clause is not expanded, and ϕ and ψ involve the same variables, then we try to expand it using the contrapositive implicative constraints

((IMPLIES $\neg\psi$ $\neg\phi$) σ_1 $\sigma_2 ... \sigma_n$).

Further Extensions

We previously mentioned our recent weakenening of the verification conditions for combined implicative and sv-constraints, making stronger use of the induction assumptions. The nature of the change is best appreciated from an example. Consider the following simple set of operators describing two ways of getting from one place to another – walking and taking a cab:

```
(define (operator walk)
 :parameters (?x ?y)
 :precondition
    (and (at ?x) (neq ?x ?y))
 :effect (and (at ?y)
             (not (at ?x))))

(define (operator take-cab)
 :parameters (?x ?y)
 :precondition
    (and (at-cab ?x) (neq ?x ?y))
 :effect (and (at-cab ?y)
             (not (at-cab ?x))))

(define (operator get-in)
 :parameters (?x)
 :precondition
    (and (at-cab ?x) (at ?x))
 :effect (and (not (at ?x))
             (in-cab)))

(define (operator get-out)
 :parameters (?x)
 :precondition
    (and (at-cab ?x) (in-cab))
 :effect (and (at ?x)
             (not (in-cab))))
```

A law that holds in this domain, whenever it holds initially, is `(IMPLIES (AT ?*X) (NOT (IN-CAB)))`. The point of interest is that verification of this law for the `walk` operator requires the inference, from the induction assumption, that `(NOT (IN-CAB))` holds in any state in which the `walk`-preconditions hold, and since this condition persists, that it also holds after a `walk`. Our current code does this verification by adding the immediate consequences of the induction assumption to the operator preconditions, and applying the weakened verification conditions (no longer requiring a *change* from `(in-cab))` to `(NOT (IN-CAB))` in `walk`). We note that this technique could also be used for simultaneous induction, i.e., we could assume multiple hypotheses in states prior to operator application, add the immediate consequences of these assumptions to the preconditions, and then apply our usual verification conditions for individual hypotheses. If the individual verifications succeed, then all the assumed hypotheses are true (*cf.* (Rintanen 1998)).

Besides the implemented extensions we have outlined so far, we have also formulated a number of extensions theoretically, which we now describe briefly. Their implementation remains as future work.

Strict Single-Valuedness (and n-Valuedness)

The verification conditions we described for "dedicated" inference of sv-constraints essentially ensure that (a) no operator application generates multiple literals that would violate the sv-constraint, and (b) any operator application that generates one instance of the predicate at issue, where there might already be a prior instance with the same "unstarred" arguments, also generates a compensating change from an instance to a negated instance of the predicate. Thus the number of tuples of values of starred variables, for any given values of the unstarred ones, is limited to 1. To establish strict singlevaluedness, we need only add the converse of (b), that any operator that produces a negated instance of the predicate at issue should also produce a "compensating" positive instance. However, both effects must also appear in negated form in the preconditions, since mere affirmation of an effect may just be reaffirmation of something that was already true in the prior state.

In other words, we verify that for any given values of the unstarred arguments, the number of tuples of values of the starred arguments remains fixed. An interesting point is that this enables discovery of "strict n-valuedness" constraints as readily as strict sv-constraints. The only difference lies in what can be confirmed in the initial state.

N-ary Disjunctive Constraints

Our implemented methods in principle allow the inference of constraints involving any number of literals – but only two of these may be fluent literals (the rest are static supplementary conditions). However, our hypothesis generation and verification techniques are rather readily extensible to arbitrary disjunctions of literals, where any number of these may be fluents. We require the hypothesis to be of form $\phi_1 \vee ... \vee \phi_n$, where the ϕ_i are literals, and for each ϕ_i for which some instances can become false (through the effects of some operator), there is another literal ϕ_j whose variables are a subset of those of ϕ_i. All variables are regarded as universally quantified. Then we can verify the law by confirming that whenever one of the literals becomes false (i.e., an instance of its negation is asserted in an operator's effects), a corresponding instance of another literal, with at most the same variables, becomes or remains true (i.e., is asserted as a persistent precondition, or as an effect). We have formulated a method that starts with (potentially) "unnecessarily lengthy" disjunctions, and systematically finds minimal combinations of literals whose disjunction is an invariant.

```
BW-LARGE-B
 ((IMPLIES (CLEAR ?X) (BLOCK ?X)))
 ((IMPLIES (CLEAR ?X) (NOT (FIXED ?X))))
 ((IMPLIES (ON ?X ?Y) (BLOCK ?X)))
 ((IMPLIES (NOT (BLOCK ?X)) (CLEAR ?X)))
 ((IMPLIES (ON ?X ?Y) (BLOCK ?Y)))
 ((IMPLIES (ON ?X ?Y) (NOT (ON ?Y ?X))) )
 ((IMPLIES (ON ?X ?Y) (NEQ ?X ?Y)))
 ((IMPLIES (ON ?X ?Y) (NOT (FIXED ?X))))
 ((IMPLIES (ON ?*X ?Y) (NOT (CLEAR ?Y))) (NOT (FIXED ?Y)))
 ((IMPLIES (FIXED ?X) (BLOCK ?X))) ((BLOCK ?X))
 ((ON ?X ?*Y)) ((XOR (ON ?X ?Y) (CLEAR ?Y)) (NOT (FIXED ?Y)))

BW-LARGE-B1
 ((IMPLIES (ON ?X ?Y) (NEQ ?X TABLE)))
 ((IMPLIES (ON ?X ?Y) (NEQ ?X ?Y)))
 ((IMPLIES (ON ?*X ?Y) (NOT (CLEAR ?Y))) (NEQ ?Y TABLE))
 ((IMPLIES (ON ?X ?Y) (NOT (ON ?Y ?X))))
 ((ON ?X ?*Y)) ((XOR (ON ?X ?Y) (CLEAR ?Y)) (NEQ ?Y TABLE))
```

Figure 1: Samples of DISCOPLAN outputs for `bw-large-b` and `bw-large-b1`.

N-ary Disjunctive and SV-Constraints

The second generalization concerns disjunctive laws with simultaneous assumption of sv-constraints. We assume a disjunction of literals $\phi_1 \vee ... \vee \phi_n$, where the ϕ_i literals may contain "starred" occurrences of some variables, subject to the following constraints: (a) only negative literals may contain "starred" variable occurrences; (b) "starred" variables may not occur in more than one literal; and (c) for every ϕ_i that can become false, the set of "unstarred" variables occurring in that literal must include the set of variables (possibly the empty set) of some other literal.

Note that (c) is a generalization of the constraint assumed above for implicative laws without "starred" variables, i.e., without sv-hypotheses. The interpretation of any "stars" occurring in a negative literal is that the predicate of that literal is single-valued in the sense that if we consider all tuples of values of the predicate that satisfy the supplementary conditions on them, then we will find at most one such tuple for any particular choice of values of the unstarred arguments.

For such a generalized disjunctive and sv-hypothesis we have formulated, and proved to be correct, verification conditions enabling an inductive proof that the law holds in all reachable states. These conditions are a generalization of those for n-ary disjunctive constraints, and implicative constraints involving non-subsumed starred variables, that we have described in the previous sections.

Sample Results and Related Work

Figures 1-2 give the outputs of DISCOPLAN for some known problems. For lack of space fluent predicate domains and OR-constraints are not included; moreover, we report only 6 of the 36 static constraints for `T-trains1` and 6 of the 15 static constraints for `att-logistics`. `logistics-a` is from the ATT-logistics domain (Kautz & Selman 1996) for which we used the formalization provided by MEDIC (Ernst, Millstein & Weld 1997); `T-trains1` is from the typed version of the Trains domain (Gerevini & Schubert 1996) containing seven operators, one of which has conditional effects; `bw-large-b` is a problem from SATPLAN formalization of the blocks world (Kautz & Selman 1996) that we translated into four operators with no conditional effects; `bw-large-b1` is the same problems as `bw-large-b`, except that here we used a different domain formalization with just one operator containing conditional effects:

```
T-TRAINS1
 ((IMPLIES (AT ?X ?Y) (CITY ?Y)))
 ((IMPLIES (IN ?*X ?Y) (NOT (EMPTY ?Y))))
 ((IMPLIES (OJ ?X) (NOT (ORANGES ?X))))
 ((IMPLIES (EMPTY ?X) (CAR ?X)))
 ((IMPLIES (COUPLED ?X ?Y) (CAR ?Y)))
 ((IMPLIES (LOOSE ?X) (CAR ?X)))
 ((IMPLIES (IN ?X ?Y) (OJ ?X)) (TANKER-CAR ?Y))
 ((IMPLIES (COUPLED ?*X ?Y) (NOT (LOOSE ?Y))))
 ((IMPLIES (COUPLED ?X ?Y) (ENGINE ?X)))
 ((AT ?X ?*Y) (ENGINE ?X))
 ((IMPLIES (AT ?X ?*Y) (NOT (IN ?X ?*Z))) (BANANAS ?X))
 ((IMPLIES (AT ?X ?*Y) (NOT (IN ?X ?*Z))) (COMM ?X))
 ((IMPLIES (AT ?X ?Y) (NOT (AT ?Y ?X))))
 ((IMPLIES (COUPLED ?X ?Y) (NOT (COUPLED ?Y ?X)))))
 ((XOR (COUPLED ?X ?Y) (LOOSE ?Y)) (CAR ?Y))
 ((XOR (AT ?X ?Y) (IN ?X ?Z)) (BANANAS ?X) (COMM ?X))
 ((XOR (AT ?X ?Y) (IN ?X ?Z)) (BANANAS ?X))
 ((XOR (IN ?X ?Y) (EMPTY ?Y)) (CAR ?Y)))
 ((IMPLIES (CAR ?X) (NOT (CITY ?X))))
 ((IMPLIES (CAR ?X) (NOT (TRACK ?X))))
 ((IMPLIES (CAR ?X) (NOT (ENGINE ?X))))
 ((IMPLIES (CAR ?X) (NOT (OJ-FAC ?X))))
 ((IMPLIES (CAR ?X) (NOT (COMM ?X))))
 ((IMPLIES (BANANAS ?X) (NOT (CAR ?X))))
 ((IMPLIES (BOXCAR ?X) (CAR ?X)))....

ATT-LOGISTICS-A
 ((IMPLIES (IN ?X ?Y) (OBJECT ?X)))
 ((IMPLIES (AT ?X ?Y) (LOCATION ?Y)) (OBJECT ?X))
 ((IMPLIES (AT ?X ?Y) (AIRPORT ?Y)) (AIRPLANE ?X))
 ((IMPLIES (AT ?X ?Y) (LOCATION ?Y)) (TRUCK ?X))
 ((AT ?X ?*Y) (AIRPLANE ?X))  ((AT ?X ?*Y) (TRUCK ?X))
 ((IMPLIES (AT ?X ?*Y) (NOT (IN ?X ?*Z))) (OBJECT ?X))
 ((IMPLIES (IN ?X ?Y) (NOT (IN ?Y ?X))))
 ((IMPLIES (AT ?X ?Y) (NOT (AT ?Y ?X))))
 ((XOR (AT ?X ?Y) (IN ?X ?Z)) (OBJECT ?X))
 ((IMPLIES (AIRPLANE ?X) (NOT (OBJECT ?X))))
 ((IMPLIES (AIRPLANE ?X) (NOT (TRUCK ?X))))
 ((IMPLIES (AIRPLANE ?X) (NOT (LOCATION ?X))))
 ((IMPLIES (AIRPLANE ?X) (NOT (AIRPORT ?X))))
 ((IMPLIES (AIRPLANE ?X) (NOT (CITY ?X))))
 ((IMPLIES (AIRPORT ?X) (LOCATION ?X)))....
```

Figure 2: Samples of DISCOPLAN outputs for `att-logistics` and `T-trains1`.

```
(define (operator Put)
  :parameters (?x ?y ?z)
  :precondition (and (on ?x ?)  (clear ?x)
                     (neq ?x Table) (neq ?y ?z) (neq ?x ?y))
  :effect (and (when (eq ?y Table)
                   (and (on ?x ?y) (clear ?z) (not (on ?x ?z))))
               (when (and (neq ?y Table) (clear ?y))
                   (and (on ?x ?y) (clear ?z) (not (on ?x ?z))
                        (not (clear ?y))))) )
```

The tests were conducted on a portable PC 266 MHz with 64 Mbytes, running Allegro Common Lisp 4.3 under Linux.[5] Each type of constraint can be discovered in isolation using a dedicated inference procedure. The total CPU-times that were required for computing all types of constraints for the problems of Figures 1 and 2 were: 0.056 CPU-seconds for `logistics-a`, 0.098 for `T-trains1`, 0.026 for `bw-large-b` and 0.086 for `bw-large-b1`.

Other approaches for the automatic inference of state invariants have been proposed, including (Kelleher & Cohn 1992; Kelleher 1996), (Fox & Long 1998) and (Rintanen 1998). A major contribution of our work in relation to this prior work and to the original version of DISCOPLAN (1998) is the allowance for conditional effects. Kelleher and Cohn's method is similar to our generation-and-test approach. However, their techniques cannot infer the types of constraints that we have addressed in this paper. The algorithm for computing state invariants proposed in (Rintanen 1998) is limited to propositional operators, and it appears to be computationally more expensive than our techniques.

Fox and Long's TIM system can infer some non-binary membership invariants that DISCOPLAN currently cannot infer. On the other hand, TIM does not handle negated preconditions, and cannot infer the following classes of DISCOPLAN constraints: antisymmetry constraints; XOR-constraints; strict n-valuedness; and implicative constraints $\phi \Rightarrow \psi$ where

- ϕ or ψ is a propositional literal, e.g., `((IMPLIES (ON-BOX ?X) (NOT (ON-FLOOR))))` in the Monkey domain;
- both ϕ and ψ are positive literals, e.g., `((IMPLIES (HASBANANAS) (HASKNIFE))` in the Monkey domain; or
- ψ is an `EQ/NEQ` conditions like the irreflexivity law `((IMPLIES (ON ?X ?Y) (NEQ ?X ?Y)))`.

[5]DISCOPLAN is written in Lisp and it accepts input domain descriptions specified using either the UCPOP or PDDL formalism.

Conclusions and Further Work

A significant aspect of the work we have reported here is the allowance for conditional effects in our mechanisms for discovering state constraints in planning domains. Another significant aspect is the greatly expanded range discovery techniques, many of which have been implemented in the new version of DISCOPLAN.

Future work includes the (easy) extension of our methods to use the initial state (in addition to the individual operators) to formulate hypothetical constraints, and the implementation of the techniques that have been developed theoretically but not yet implemented.

Acknowledgments

This research was supported in part by NATO grant CRG951285, and by ARPA/SSTO grant F30602-95-1-0025 DARPA grant F30602-98-2-0133 (second author). We would like to thank Piergiorgio Bertoli for a fruitful discussion that inspired our method of inferring XOR-constraints.

References

Ernst, M.; Millstein, T.; and Weld, D. 1997. Automatic SAT-compilation of planning problems. *Proc. of IJCAI-97*, 1169-1176.

Fox, M. and Long, D. 1998. The Automatic Inference of State Invariants in TIM. *JAIR*, 9, 367–421.

Fox, M. and Long, D. 2000. Utilizing Automatically Inferred Invariants in Graph Construction and Search. *Proc. of AIPS-00*.

Gerevini, A. and Schubert, L.K. 1996. Accelerating partial-order planners: Some techniques for effective search control and pruning, *JAIR*, 5, 95–137.

Gerevini, A. and Schubert, L.K. 1998. Inferring state constraints for domain-independent planning. *Proc. of AAAI-98*, 905–912.

Gerevini, A. and Schubert, L.K. 2000. Extending the Types of State Constraints Discovered by DISCOPLAN. *Proc. of the AIPS-00 Workshop on Analysing and Exploiting Domain Knowledge for Efficient Planning*.

Kautz, H., and Selman, B. 1996. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proc. of AAAI-96*.

Kautz, H., and Selman, B. 1998. The Role of Domain Specific Knowledge in the Planning as Satisfiability Framework. search. In *Proc. of AIPS-98*.

Kelleher, G. and Cohn, A.G. 1992. Automatically synthesising domain constraints from operator descriptions. *Proc. of ECAI-92.*.

Kelleher, G. 1996. Determining General Consequences of Sets of Actions. TR CMS.14.96. Liverpool Moores Univ.

Refanidis, I. and Vlahavas, I. 2000. Exploiting State Constraints in Heuristic State-Space Planning. *Proc. of AIPS-00.*

Rintanen, J. 1998. A planning algorithm not based on directional search. *Proc. of KR'98*, 617–624.