Computing Domain Invariants for Planning in DISCOPLAN*

Alfonso Gerevini
Dipartimento di Elettronica per l'Automazione
Università di Brescia, via Branze 38, I-25123 Brescia, Italy

Lenhart Schubert
Department of Computer Science
University of Rochester, Rochester, NY 14627-0226.

1 Introduction

State invariants (or *state constraints*) in planning are properties of objects or relationships among objects that hold in all states reachable from the initial state. For example, a familiar invariant in a blocks world is the property that that if one block is on another, the latter is not clear. In our terminology, this is an *implicative* constraint. Another example is that a block can be on at most one other block; this is a *single-valuedness* constraint (sv-constraint).

A point that has become widely recognized in the planning community (and that we amplify in what follows) is that knowledge of state invariants is important for efficient planning. However, such knowledge cannot in general be assumed to be available *a priori* in a given planning domain. Rather, planning domains are generally considered fully specified once a set of operators with well-defined preconditions and effects has been supplied, along with an initial state. This is defensible since state invariants are implicit in the specification of the operators and initial state; i.e., under a STRIPS assumption the only properties and relationships that change when an operator is applied are those spelled out in the effects of the operator. So a separate specification of what remains unchanged when operators are applied would be logically redundant. However, it is far from obvious from inspection of a given set of planning operators and an initial state what the invariants of the domain are. The goal of our research has been to formulate automatic, efficient methods for inferring the most important such invariants, and to implement these methods in our DISCOPLAN system.

The importance of state invariants for efficient planning is that they can be used to radically restrict the search space. This is so for any approach to planning that involves explicit or implicit exploration of incompletely specified possible states of the world, as is the case for deductive planning, regression planning, bidirectional planning, and planning by incremental constraint satisfaction (in particular, SAT-based planning). In our work we have focused on SAT-based planners. These implicitly search a space of state sequences, constrained by disjunctions of ground literals. Their performance depends critically on the invariants added (as ground instances) to the mix of disjunctions, and intuitively this is because state invariants constrain the alternative states that are possible at each time step under consideration. Some results showing the dramatic improvements in the performance of SAT-based planners like SATPLAN [6] and MEDIC [2] obtainable through the use of automatically inferred invariants are included in [3].

*Contact author: Alfonso Gerevini, email: gerevini@ing.unibs.it. DISCOPLAN is written in Common Lisp and was tested using Allegro Common Lisp. The demo of the system can be remotely run using Netscape on any hardware platform.

DISCOPLAN finds a variety of different types of constraints. The majority of its algorithms are devoted to the discovery of state invariants, using a *hypothesize-and-test* paradigm. All the algorithms instantiating this paradigm are applicable to sets of operators conforming with UCPOP or PDDL syntax [7, 5], allowing for *when*-clauses (conditional effects) but not disjunctive or universally quantified conditions. We will be referring to the unconditional part of an operator as its *primary when-clause*. The allowance for conditional effects is a major distinction of DISCOPLAN from related systems.

Very briefly, the hypothesize-and-test paradigm consists of hypothesizing invariants Γ of some particular syntactic type, such as implicative constraints (`IMPLIES` ϕ ψ) where ϕ and ψ are literals that may contain universal variables, augmenting these hypotheses with potential supplementary static conditions, and then testing them against all *when*-clauses of all operators and against the given initial conditions. In the testing phase, minimal sets Σ of supplementary conditions are found, up to sets of some limited size (e.g., 2 or 3) that suffice to ensure that $\Sigma \Rightarrow \Gamma$ holds in all states reachable from the initial state. The hypothetical invariants Γ of a particular type are chosen by inspecting the preconditions and effects of particular operators, to find conditions that appear to become or remain true when certain kinds of effects occur. The idea is to choose the constituents of Γ in such a way that a proof by induction of the invariance of Γ will be at least locally enabled. In this way large numbers of syntactically possible invariants are eliminated from consideration. The testing phase can be viewed as an automated inductive proof attempt (with addition of supplementary conditions as needed to allow the proof to succeed). An important point is that Γ may actually consist of multiple hypotheses that can be proved to be invariants by simultaneous induction. Typically, such multiple hypotheses consist of an implicative hypothesis (`implies` ϕ ψ) along with sv-hypotheses corresponding to argument positions in ϕ and ψ occupied by universal variables occurring in only one of ϕ, ψ. The point is important since the invariance of the individual formulas in such cases cannot be proved in isolation. Our various hypothesize-and-test algorithms have been proved to yield correct invariants, and run in polynomial time for fixed bound on the number of supplementary conditions Σ added to Γ.

In a little more detail, the hypothesize-and-test algorithms conform with the following structure (iterating over all possible candidate constraints Γ found in the first step).

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 its 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, e.g., 3) 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.

2 Types of Invariants Discovered by DISCOPLAN

The input of DISCOPLAN is a domain description consisting of the specification of an initial state and a set of extended STRIPS operators which may involve conditional effects, negated preconditions, constants, typed and untyped parameters. In the following we describe the types of invariants that are discovered by the current version of DISCOPLAN (for a more detailed description the reader is referred to [3, 4]).

In Figure 1 we give the invariants computed for the problem `logistics-a` from the ATT-logistics domain [6] for which we used the formalization provided by MEDIC [2] (predicate domain constraints are not included).[1]

Predicate Domain Constraints

Predicate domain constraints are sets of possible argument tuples corresponding to each predicate in the domain after 0, 1, ..., t actions have occurred. These constraints are computed using a simplified version of the planning graph for the given problem [1].

Static Predicates and Static Constraints

Static constraints are invariants involving type-predicates, i.e., static monadic predicates that occur positively in the initial state – ones unaffected by any operator. Static constraints consist of a (possibly empty) set of objects for each type-predicate and a list of supertype, subtype, and incompatible relationships between type-predicates.

Simple Implicative Constraints

Simple Implicative Constraints are constraints of form $((\phi \Rightarrow \psi)\ \sigma_1 ... \sigma_k)$, where ϕ, ψ, and $\sigma_1, ..., \sigma_k$ are function-free *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 ϕ then ψ, *provided that* σ_1, ..., and σ_k". We assume that the variables occurring in ϕ include all those occurring in ψ and in the supplementary conditions $\sigma_1, ..., \sigma_k$. The predicate in ϕ is a fluent predicate, while ψ may be fluent or static. However, if ϕ contains variables that do not occur in ψ, then ψ is required to be "upward monotonic", in the sense that no instances of it can become false ($\neg\psi$ does not unify with effect of any operator; this is certainly true if ψ is static). Finally, we require $\sigma_1, ..., \sigma_k$ to be static. The following is an example of this type of constraint in the blocks world stating that the table cannot be on any block: $\forall x, y\ ON(x, y) \Rightarrow NEQ(x, TABLE)$.

Single-valuedness Constraints

An sv-constraint states that the value of a certain predicate argument is unique for any given values of the remaining arguments. An example of an sv-constraint is the following blocks-world constraint stating that any object can be `ON` at most one other object:

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

Implicative Constraints + Single-Valuedness Constraints

These invariants are formed by an implicative constraint and a set of sv-constraints that are simultaneously discovered by DISCOPLAN. We distinguish two cases which require different verification conditions: the case of subsumed variables and the case of non-subsumed variables. The blocks-world constraint `((IMPLIES (ON ?*X ?Y) (NOT (CLEAR ?Y))) (NEQ ?Y TABLE))` is an example of a combined implicative and sv-constraint for the first case. In general, the implicative constraints we are considering here have as their antecedent a positive literal that contains at least one

[1] DISCOPLAN computes these invariants in 0.045 CPU-seconds using a portable PC 266 MHz with 64 Mbytes running Allegro Common Lisp 4.3 under Linux.

Static Predicates:
```
AIRPLANE AIRPORT TRUCK LOCATION CITY IN-CITY OBJECT
```

Static Constraints:
```
Forall x. City(x) => ~Location(x)
Forall x. City(x) => ~Truck(x)
Forall x. City(x) => ~Object(x)
Forall x. Airport(x) => ~City(x)
Forall x. Airport(x) => Location(x)
Forall x. Airport(x) => ~Truck(x)
Forall x. Airport(x) => ~Object(x)
...
```

Simple Implicative Constraints
```
Forall x,y. At(x,y)&Airplane(x) => Airport(y)
Forall x,y. At(x,y)&Truck(x) => Location(y)
Forall x,y. At(x,y)&Object(x) => Location(y)
Forall x,y. In(x,y) => Object(x)
```

Exclusive Constraints:
```
Forall x,y,z. At(x,y)&Object(x) => ~In(x,z)
```

Single-Valuedness Constraints:
```
Forall x,y,v0. At(x,y)&At(x,v0)&Airplane(x) => y=v0
Forall x,y,v0. At(x,y)&At(x,v0)&Truck(x) => y=v0
Forall x,y,z,v0. At(x,y)&At(x,v0)&Object(x) => y=v0
Forall x,y,z,v0. In(x,z)&In(x,v0)&Object(x) => z=v0
```

Antisymmetry Constraints:
```
Forall x,y. At(x,y) => ~At(y,x)
Forall x,y. In(x,y) => ~In(y,x)
```

OR Constraints:
```
Forall x. Exist y,z. Object(x) => At(x,y) OR In(x,z)
```

XOR Constraints:
```
Forall x. Exist y,z. Object(x) => At(x,y) XOR In(x,z)
```

Figure 1: Invariants for `logistics-a` (only 7 of the 15 static constraints are reported)

"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.

In the second case we have implications in which 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))) (OBJECT ?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, and `IN` no more than one thing).

Antisymmetry Constraints

Antisymmetry constraints are particular implicative constraints of the form ((`IMPLIES` $(P\ t_1\ t_2)$ (`NOT` $(P\ t_2\ t_1)$)) $\sigma_1\ \sigma_2 ... \sigma_n$), 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 $\forall x, y.\ ON(x, y) \Rightarrow \neg ON(y, x)$, i.e., if one object is on another, then the second is not on the first.

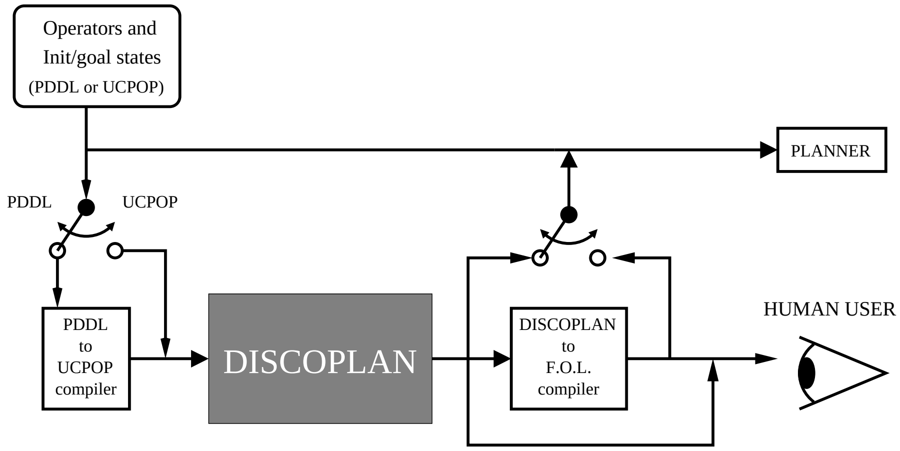

Figure 2: General scheme of DISCOPLAN's input/output

OR and XOR constraints

OR and XOR-constraints are state constraints of the form ((`[X]OR` 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.

Strict Single-Valuedness and n-Valuedness Constraints

This type of invariant is a generalization of sv-constraints. A *nv*-constraint states that a certain predicate can be bound to at most n arguments for any given values of the remaining arguments. A strict nv-constraint states that a certain predicate is bound to exactly n arguments for any given values of the remaining arguments. An example of a strict nv-constraint with $n = 1$ in the blocks world is the invariant stating that any block is on exactly one thing (either another block or the table).

Using "Expanded Operators" to Infer Further Constraints

DISCOPLAN's package includes routines for expanding an operator with a set of given invariants. The operator expansion consists of enriching the operator description with additional preconditions and effects that are entailed by the given invariants. By using expanded operators DISCOPLAN may infer new invariants, which can be used to expand the operators again. This can be iterated until no new constraints are inferred.

3 DISCOPLAN input/output

The general input/output scheme of DISCOPLAN is described in Figure 2. The input domain and problem descriptions can be specified using the syntax of either UCPOP or PDDL. Since the core functions of DISCOPLAN assume UCPOP descriptions, when the input is specified using PDDL, it is automatically translated into a UCPOP set of operators.

The output of DISCOPLAN can be given as input to either a planner that can exploit this information, or to a domain developer, as an aid to domain specification and debugging. The syntax of the output can be either FOL or the compact format using "starred" variables that we used in the previous sections. This second format is more compact than the FOL format because an implicative, or exclusive constraint that is discovered simultaneously with sv-constraints is written as a single constraint, corresponding to multiple constraints in the FOL format. The FOL description of the state constraints is obtained by a postprocessing step translating the constraints computed in DISCOPLAN format into FOL.

References

[1] A. Blum and M.L. Furst. Fast planning through planning graph analysis. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 1636–1642. Morgan Kaufmann, 1995.

[2] M.D. Ernst, T.D. Millstein, and D.S. Weld. Automatic SAT-compilation of planning problems. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 1169–1176. Morgan Kaufmann, 1997.

[3] A. Gerevini and L. Schubert. Inferring state constraints for domain-independent planning. In *Proceedings of the Fifteenth National Conference of the American Association for Artificial Intelligence (AAAI-98)*, pages 905–912. AAAI Press/The MIT press, 1998.

[4] A. Gerevini and L. Schubert. Inferring state constraints in DISCOPLAN: Some new results. In *Proceedings of the Seventeenth National Conference of the American Association for Artificial Intelligence (AAAI-00)*, pages 761–767. AAAI press / The MIT Press, 2000.

[5] M. Ghallab, A. Howe, G. Knoblock, D. McDermott, A. Ram, M. Veloso, D. Weld, and D. Wilkins. PDDL – planning domain definition language. Technical report, 1998. Available at `http://cs-www.cs.yale.edu/homes/dvm/`.

[6] H.A. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the Thirteenth National Conference of the American Association for Artificial Intelligence (AAAI-96)*, Portland, OR, 1996.

[7] J.S. Penberthy and D.S. Weld. UCPOP: A sound, complete, partial order planner for ADL. In *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning (KR'92)*, pages 103–114, Boston, MA, 1992. Morgan Kaufmann.