Discovering State Constraints for Planning with Conditional Effects in DISCOPLAN (Part I)*

Alfonso Emilio Gerevini[1] and Lenhart Schubert[2]

[1] *Dipartimento di Ingegneria dell'Informazione, Università di Brescia*
Via Branze 38, 25123 Brescia, Italy. E-mail: alfonso.gerevini@unibs.it

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

Corresponding author: Alfonso E. Gerevini (alfonso.gerevini@unibs.it)

Abstract

DISCOPLAN is a durable and efficient system for inferring state constraints (invariants) in planning domains, specified in the PDDL language. It is exceptional in the range of constraint types it can discover and verify, and it directly allows for conditional effects in action operators. However, although various aspects of DISCOPLAN have been previously described and its utility in planning demonstrated, the underlying methodology, the algorithms for the discovery and inductive verification of constraints, and the proofs of correctness of the algorithms and their complexity analysis have never been laid out in adequate detail. The purpose of this paper is to remedy these lacunae.

1 Introduction

State constraints, also called state invariants, have long been known to be useful in reducing planning complexity by enabling avoidance of unreachable states. Even Genesereth and Nilsson's classic book on logical foundations of artificial intelligence pointed out that state constraints enabled "unachievability pruning" in deductive planning (see [15], section 2.9), and planning techniques developed since then (especially those employing satisfiability testing or related techniques) make crucial use of state constraints for efficient plan generation. The importance of computing and exploiting state constraints in planning has been reiterated many times; recent examples are [1, 30].

While many useful state constraints can be "intuited" via human understanding of a planning domain, manual axiomatization is an undesirable and unnecessary burden, if constraints can be automatically inferred. Moreover, it is well-known in the planning community that errors in operator specifications occur all too easily, and automated constraint

*This article revises and significantly extends material reported in several previous conference and workshop papers [19, 21, 20, 22] – generally at a rather sketchy and abstract level, as necessitated by these venues. The most important contribution here is the detailed, unified presentation of the essential ideas and algorithms underlying DISCOPLAN and their soundness proofs and complexity analyses.

inference often reveals such "bugs" quickly by bringing to light the bizarre constraints they engender. Thus, inference of state constraints becomes an increasingly important debugging tool as planning domains are scaled up from toy problems to real-world problems.

In this paper we describe DISCOPLAN, one the first and most successful systems for inferring and using state invariants. It is cited by many researchers in the field (e.g., [1, 3, 7, 8, 9, 10, 12, 14, 24, 25, 28, 29, 30, 35, 38, 39, 40]), and it contributed to the growth of interest in this important research topic in automated planning. Over the years various other systems have been proposed, some of them using and developing quite similar ideas, such as DISCOPLAN's hypothesize-and-test paradigm. DISCOPLAN is also the invariant inference system that supports the largest set of types of invariants for non-quantitative domains, including some types that no other system infers.

A comprehensive treatment of the full range of methods, algorithms, theoretical results, and experimental evaluations that have been developed for DISCOPLAN over the years is beyond the scope of a single paper. Aspects of DISCOPLAN covered in previously published work include the use of parameter domain inference to speed up planning [18], the basic idea behind our method of deriving constraints, with demonstration of their utility in SAT-planning [19], and extensions of our techniques to allow for conditional effects, antisymmetry constraints, OR and XOR contraints, as well as other types of constraints and some methods of bootstrapping [21, 20, 22]. However, no detailed description of the algorithms and methodology, proofs of correctness, and complexity analyses have been provided before, and the goal of this paper is to provide these important details.

We will focus on our most essential methods and on the basic types of state constraints, on which additional inference methods and specific techniques are built. These techniques are at the core of DISCOPLAN, and generally suffice for efficient inference of the great majority of the invariants discovered by the full DISCOPLAN system. The interested reader can find a description of the additional methods and techniques, together with an empirical evaluation of the whole system in [19, 21, 16].

1.1 Related Work

Many approaches to infer and use state invariants have been proposed for planning [5, 8, 9, 13, 17, 27, 26, 14, 25, 29, 30, 32, 36, 37, 38, 41], and more recently also for temporal planning [25, 39]. While a detailed comparison between DISCOPLAN and all other existing systems for automatic inference of state invariants is outside the scope of this paper, in the following we briefly outline the main differences. (The interested reader can see [16] for an in-depth comparison with several other systems.)

Most of the related work relies on techniques similar to our hypothesize-and-test paradigm, with specific differences in the ways hypotheses are generated, weakened, and tested; an exception is Fox and Long's TIM system, which uses a hierarchy of inferred object types containing sets of functionally equivalent objects.

Another major difference concerns the syntactic restrictions on the types of inferred invariants. For instance, in TIM formulas ϕ and ψ of an implicative constraint $\phi \Rightarrow \psi$ cannot involve more than one universally quantified variable, cannot be propositional literals, must both be positive without constant arguments, and moreover ψ cannot be an `EQ/NEQ`-condition; in Rintanen's system [36, 37] supplementary conditions involving constants are not generated (e.g., `((IMPLIES (ON ?X ?Y) (NOT (CLEAR ?Y))) (NEQ ?Y TABLE))`; Helmert's system [25] focuses on mutex invariants. Moreover, some of the existing systems infer

only grounded invariants, while DISCOPLAN discovers first-order invariants (contrary to comments in [7]), and no other system infers DISCOPLAN's type constraints.

Concerning the supported planning language, DISCOPLAN can directly deal with action models (operators) involving some features that other systems can handle only by precompiling them away [4, 33]. These are `EQ/NEQ`-conditions, negated preconditions, constants, and especially conditional effects. In practice compilation of conditional effects can be an acceptable solution when they are very few, but in general it leads to an exponential blow-up of the domain actions, or to a polynomial increase in the plan size, which can make plan generation substantially harder or even infeasible.

The only other approach supporting conditional effects is Helmert's, which adopts a method similar to our hypothesize-and-test paradigm with a different method for refining hypothesized invariants and verifying them. This technique was designed primarily to derive invariants whose use is helpful in building concise finite-domain representations in the translation of the planning problem specification from PDDL to SAS$^+$ [6] that is used in the context of the Fast Downward planner [25]. The translation uses invariants corresponding to mutually exclusive atomic propositions, stating that at most one of such propositions can hold in any reachable state. A set of propositions that are pairwise mutually exclusive can be encoded as a single state SAS$^+$ variable whose value specifies which of the propositions is true (or that none of them is true at all). DISCOPLAN was designed to be a general tool and discovers more types of invariants. In particular, with respect to the class of DISCOPLAN's invariants investigated in this paper, Helmert's system does not infer implicative constraints where both ϕ and ψ are positive or ψ is a persistent precondition (one that remains true when the operator is applied), sv-constraints where more than one variable is starred, and some constraints involving constant symbols and `EQ/NEQ`-conditions. Moreover, we observed that for some benchmark domains DISCOPLAN derives some mutex invariants that are not found by Helmert's technique.

On the other hand, other systems have their own advantages, and in general there is no system or approach that universally outperforms all others in terms of the number of inferred invariants, the computation time to derive them, or features of the planning language that are supported. For instance, TIM is often very fast; Rintanen's approach is general, relatively simple and elegant, but slower than others (although recent efforts have been dedicated to improving efficiency [40]); Helmert's approach supports universally quantified variables in preconditions and effects (DISCOPLAN makes only very limited allowance for such variables), and since it focuses on a particular class of invariants, it can be much more efficient for these invariants, especially with large PDDL domain specifications. Finally, we should mention some additional types of constraints (not discussed here), most of which only DISCOPLAN is capable of discovering, namely OR-XOR constraints, strict sv-constraints, n-valuedness constraints for $n > 1$, n-ary disjunctive constraints for arbitrary positive n, and the latter in conjunction with sv-constraints.

1.2 Overview

The technical presentation of our constraint discovery techniques begins in section 2 with the assumed representations for planning domains and domain constraints, and a preliminary technique for discovering *type constraints*, i.e., monadic constraints on objects in the initial state that are not affected by any operator. For example, in a transportation domain certain objects might be designated as vehicles and more specifically as trucks, airplanes,

or trains, others as locations (such as cities), others still as particular kinds of cargo, etc. Often these properties are related hierarchically (for instance, all trucks are also vehicles) and by exclusion relations (for instance, no truck is an airplane).

In section 3, we motivate and present our hypothesize-and-test paradigm for discovering a variety of constraints "hinted at" by operator precondition-effect structure. For example, in a traditional blocks world the `Put` operator has effects `(ON ?x ?y)` and `(NOT (CLEAR ?y))` (when `?y` is not the `TABLE`), suggesting the constraint that whenever `(ON ?X ?Y)` holds, so does `(NOT (CLEAR ?Y))` (when `?Y` is not the `TABLE`).[1] Verifying such a potential state constraint requires an inductive proof that if the constraint holds in a given state, then it still holds in any successor state that can be generated by an operator application; and as basis case, the constraint needs to be verified in the initial state. In order to efficiently implement the inductive proof step, we proceduralize appropriate *verification conditions*, aimed at specific types of constraints. For example, for the implicative constraint mentioned above, we need to be sure that whenever the antecedent `(ON ?X ?Y)` is made true by some operator for some `?X` and `?Y` (where `?Y` is not the `TABLE`), the consequent `(NOT (CLEAR ?Y))` is also made true by that operator, or persists from the prior state. In addition, we need to be sure that whenever `(NOT (CLEAR ?Y))` is made false (i.e., `(CLEAR ?Y)` is made true) by some operator for some `?Y` (where `?Y` is not the `TABLE`), then `(ON ?X ?Y)` is also made false *for all* `?X`, or this condition persists from the prior state. (Note that if it can be shown simultaneously that for any given `?Y`, `(ON ?X ?Y)` holds for *at most one* `?X`, then proving that `(ON ?X ?Y)` becomes false for *some* `?X` is sufficient to show that it becomes false for *all* `?X`. This observation provides a preliminary glimpse of the way in which *single-valuedness constraints* interact with implicative constraints – an important theme in much of what follows.) In general, formulating verification conditions for a given type of constraint is a matter of considering all the ways a constraint of that type could be falsified by an operator application (when it was true in the prior state), and positing sufficient conditions on the preconditions and effects of operators to ensure that no such violation can occur.

After stating the generic algorithm we employ for hypothesizing and verifying constraints (subsection 3.1), we discuss in more detail how verification conditions, supporting the inductive proofs of hypothetical constraints, are formulated (subsection 3.2), how potential supplementary conditions are collected (subsection 3.3) how irredundant subsets are selected from them (subsection 3.4), and how constraints and their potential supplementary conditions are filtered through the initial state (subsection 3.5). The most subtle aspect is the collection of supplementary conditions, which serve to "excuse" apparent violations of the verification conditions. For example, coming back to the previous blocks-world illustration, one hypothesis based on the `Put` operator would just be that `(ON ?X ?Y)` implies `(NOT (CLEAR ?Y))`. The corresponding verification conditions would be found to be violated by the `Put` operator when the value of the parameter `?y` corresponding to variable `?Y` is the `TABLE`, since in that case no effect `(NOT (CLEAR ?y))` is asserted even though `(ON ?x ?y)` is asserted. But our algorithms would take note of the precondition `(EQ ?y TABLE)` as a possible "excuse" in that case, and would ultimately "rescue" the implicative constraint (in conjunction with a single-valuedness constraint) by adding `(NOT (EQ ?Y TABLE))` as a supple-

[1]Note that we use lower-case symbols such as `?x` and `?y` for operator parameters, and upper-case symbols such as `?X` and `?Y` for implicitly universal variables in state constraints. Also, we generally write operator names with an initial capital (e.g., `Put`), and use uniform upper case for predicates, constants and logical connectives (`AND, OR, NOT, IMPLIES`); however, for computer-interpretable operator specifications we assume case-insensitivity (and prefer uniform lower case for compactness of presentation).

mentary condition. In the final subsection (3.6), we provide some methodological remarks on the pros and cons of bootstrapping in hypotheses testing.

In section 4, we present our basic instantiations of the hypothesize-and-test paradigm, considering implicative and single-valuedness constraints first in isolation and then in combination. In particular, in section 4.1 we outline the discovery of "simple" implicative constraints, as an instance of our hypothesize-and-test paradigm. These are ones that do not require simultaneous consideration of single-valuedness constraints. An example is the TRAINS world constraint that when `?X` is `AT ?Y`, then `?Y` is a `CITY` [2, 18]. An example of a single-valuedness constraint (sv-constraint) that can be discovered in isolation (section 4.2) is the "logistics" world constraint that `?X` can be `AT` only one `?Y`, whenever `?X` is an `AIRPLANE` (similarly, if `?X` is a `TRUCK`). An example of a combined implicative and sv-constraint (section 4.3) is the blocks-world constraint mentioned above, or the TRAINS world constraint that if `?Y` (e.g., a boxcar) is `COUPLED` to `?X` (an engine), then `?Y` is not `LOOSE`, and furthermore, it is not `COUPLED` to anything else (single-valued in `?Y`).

In contemplating the details (in section 4) of our various instantiations of the hypothesize-and-test paradigm, one may well wonder why separate verification conditions and algorithms are needed for each class of invariants. The answer is that distinct methods of mathematical induction, and hence distinct algorithmic techniques, are needed for each class. Unfortunately, we have no uniform, efficient proof procedures for reasoning about dynamic worlds by mathematical induction, incorporating a closed world assumption and unique names assumption (see below), and capable of handling equalities and inequalities. Thus, some human ingenuity is required to formulate verification conditions that will enable inductive proofs to succeed for most of the cases of interest, and to write code that efficiently and correctly implements the induction steps in each case.

2 Preliminaries

In this section we describe the assumed representation for planning domains and state constraints, introduce some notation, and propose a technique for computing type constraints.

2.1 Representation of Planning Domains and State Constraints

DISCOPLAN handles both PDDL [23] and UCPOP-style [34] operators. Disjunctive conditions are not handled. Universal quantification as well is disallowed by most of the specialized routines implemented in DISCOPLAN. These features of the planning language can be compiled away [33], and in [16] additional techniques that support them are presented.

The following is a PDDL encoding of a blocks-world `Put` operator illustrating most of the features we allow for (equality and inequality conditions are indicated with the slightly more compact UCPOP-style notation "`eq`" and "`neq`", respectively).

```
(action Put
  :parameters (?x ?y ?z)
  :precondition (and (on ?x ?z) (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))))) )
```

Note the conditional effects signalled by the *when*-clauses. These serve here to distinguish the preconditions and effects of the operator when the destination `?y` of a `Put` action is the `Table`, and when it is not – where in the latter case, `?y` must initially be `clear` for the action to have any effect, and is not `clear` at the end. In general, the keyword `:effect` may be followed by a conjunction of unconditional (primary) effects and conditional (*when*)-clauses, where each conditional clause supplies some further preconditions and corresponding effects. DISCOPLAN converts such operators into a standardized, conjunction-free form. A standardized 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 assume that operators are consistent, i.e., they never produce two contradictory effects. Also, they do not have contradictory preconditions within the same *when*-clause.[2]

We assume (as is done in PDDL) that the initial state is exhaustively specified through a set of ground predications, i.e., positive predicate instances with constant arguments. Thus any ground predication not found in the initial state specification is taken to be false (a simple closed world assumption (CWA)). `EQ`- (equality-) and `NEQ`- (nonequality-) formulas are an exception to this. All formulas of form `(EQ` c c`)` and `(NEQ` c c'`)`, where c and c' are distinct constants, are taken to be implicitly true (and correspondingly, `(EQ` c c'`)` and `(NEQ` c c`)` are taken to be false). This amounts to a simple (nondeductive) *unique names assumption* (UNA).

In general we will take a *state constraint*, or *invariant*, to be any formula of function-free first-order logic restricted to the vocabulary (individual and predicate constants) of a particular planning domain, where that formula is true in all states reachable from a given initial state. Individual constants and variables are interpreted as ranging over the objects of the planning domain, but not over states or times. Thus state-dependence of truth values is implicit. Of course, we are interested in contingent formulas rather than ones like $(\forall x)\mathit{CLEAR}(x) \vee \neg \mathit{CLEAR}(x)$ that are logically true and thus provide no domain-dependent information.

A formal understanding of the preceding remarks requires a definition of *reachable states*, given an initial state in a planning domain, and this in turn depends on the definition of *states, operator instances*, and the *resulting state* when an operator instance is applied to a state. For readers not familiar with these notions, we provide informal explanations here. More formal definitions and further clarifications can be found at the beginning of Appendix A.

A *planning domain* consists of a set of PDDL operators with the structure described above. All preconditions and effects are positive or negated predications over constants and operator parameters (but `EQ/NEQ`-preconditions occur in positive form only). A *state*

[2]One point not illustrated by the above example is that some or all parameters may be typed, as in `:parameters (?x - block ?y ?z)`; in the standardized form of operators, such type information is amalgamated into the primary preconditions. (DISCOPLAN decides for itself which properties are type properties – which may include ones not explicitly supplied as types of operator parameters.)

of a planning domain is a finite set of ground atoms, exclusive of `EQ/NEQ` predications. `(EQ` c c`)` and `(NEQ` c c'`)` (with c, c' distinct) are regarded as tacitly present in every state. An *instance* of an operator is obtained by grounding its parameters (substituting constants for them), without thereby creating faulty `EQ/NEQ` preconditions, i.e., ones of type `(NEQ` c c`)` or `(EQ` c c'`)` with c, c' distinct.

The preconditions of a (primary or secondary) *when*-clause of an operator instance are *satisfied* by a state if the positive preconditions of the instance occur in the state, and the negative preconditions do not occur in positive form in the state. (Avoidance of faulty `EQ/NEQ` preconditions is guaranteed by the definition of an operator instance.) Evidently the instance is applicable to a state iff its primary preconditions are satisfied by the state. The new state resulting from applying an operator instance to a given state is obtained by implementing the positive and negative effects of the primary *when*-clause and of each secondary *when*-clause whose preconditions are satisfied. Implementing a positive effect means adding it to the state, and implementing a negative effect means removing the positive version from the state. Operators are assumed to be formulated so that the ordering of additions and deletions doesn't matter.

Finally, a *reachable state* in a planning domain, for a given initial state, is one that can be generated by a succession of operator instances, where each instance is applicable in the state where it is applied, and its result state in turn satisfies the primary preconditions of the next instance, or (for the final instance) is the reachable state in question.

2.2 Notational Variants

We will generally use abbreviated, computer-oriented ways of writing state constraints. (DISCOPLAN provides output options allowing for both the abbreviated notation and FOL notation.) In particular, we omit quantifiers in the abbreviated form, and use Lisp-like bracketing and prefixing. For example, the blocks-world constraint mentioned earlier, that when an object `?X` is `ON` an object `?Y`, then `?Y` is not clear, provided that `?Y` is not the `TABLE`, would be written as follows:

```
((IMPLIES (ON ?X ?Y) (NOT (CLEAR ?Y))) (NEQ ?Y TABLE)).
```

Note that we add supplementary conditions as a trailing list to the main implication (conditional formula); they could of course be conjoined into the antecedent of the implication, but we keep them separate because of the distinct methods by which they are obtained. Written out in FOL, this state constraint is equivalent to either of the following (where in the second formulation we allow for reverse implication, '$\Leftarrow$'):

$\forall x, y.(ON(x, y) \wedge \neg(y = TABLE)) \Rightarrow \neg CLEAR(y)$;
$\forall x, y.((ON(x, y) \Rightarrow \neg CLEAR(y)) \Leftarrow \neg(y = TABLE)$.

We abbreviate sv-constraints by using "starred" variables, as in the following example from the TRAINS domain: `((AT ?X ?*Y) (ENGINE ?X))`. This states that whenever `?X` is an engine, there can be at most one value of `?Y` such that `?X` is `AT ?Y`; or in FOL notation,

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

In general, multiple argument positions of a predicate may be "starred", indicating that at most one tuple of values may fill those positions, given any particular tuple of values in the remaining variable argument positions, and given that the supplementary conditions (if any) hold. The "star" notation can also be used in the context of implicative constraints,

again indicating single-valuedness under the assumption that the supplementary conditions hold. For instance, in the first blocks-world example above, we could replace `(ON ?X ?Y)` by `(ON ?*X ?Y)` to indicate that `ON` is single-valued in its first argument, for all pairs of arguments where `(NEQ ?Y TABLE)` holds.

2.3 Type Constraints

We define *static predicates* as ones that do not occur in any operator effects; thus static predications that are true (or false) in the initial state are true (or false, respectively) in all reachable states. Nonstatic predicates are termed *fluent* predicates. Type constraints are state invariants involving *type-predicates*, where a type-predicate is a static monadic predicate that occurs positively in the initial state.[3] 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);
- a list of *supertype/subtype* and *incompatible* relationships between type-predicates.

Type information is computed in polynomial time by an algorithm called *Find-type-constraints* that is given in Figure 1. *Find-type-constraints* processes 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, and can be found in $O(|s_0| + |\mathit{effects}|)$ time, where $|s_0|$ is the size of the the initial state specification and $|\mathit{effects}|$ is the size of all operator effects taken together). 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.[4] This set, indicated by $\|P\|$, is the *extension* of P. If we have $\|P\| = \Theta$, 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 $\|P\| \subseteq \|Q\|$, 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; finally, 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 predications 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))`}

[3] We assume that all type predicates appear (positively) in the initial state. If there are no such instances, then any negative type predication in an operator precondition is redundant, and any positive type predication in an operator precondition invalidates the operator.

[4] Here we assume that all the objects (constant symbols) of the domain appear in the initial state as terms of some positive literal. If this is not true in the original formalization, we can augment the initial conditions with dummy predications (using dummy predicates not occurring elsewhere) introducing those constants.

Algorithm: *Find-type-constraints(I)*

I: an initial state (list of positive ground literals)

Output: lists of type-constraints and of universal types

1. Initialize set of constants Θ, *type-preds*, *universal-types*, and *type-constraints* to $\emptyset$;
2. **for** each positive predicate instance p in the initial state I
3. add the constants in p to $\|Pred(p)\|$ and to Θ;
4. **if** $Pred(p)$ is static and monadic **then** add $Pred(p)$ to *type-preds*;
5. **for** each predicate name P in *type-preds* **do**
6. **if** $\|\mathrm{P}\| = \Theta$ **then** add P to *universal-types*, and `(P ?X)` to *type-constraints*;
7. **for** each predicate name Q in *type-preds* s. t. P is lexically prior to Q **do**
8. meet := $\|\mathrm{P}\| \cap \|\mathrm{Q}\|$;
9. **if** meet = $\emptyset$ **then** add `(IMPLIES (P ?X) (NOT (Q ?X)))` to *type-constraints*;
10. **if** meet = $\|\mathrm{P}\|$ **then** add `(IMPLIES (P ?X) (Q ?X))` to *type-constraints*;
11. **if** meet = $\|\mathrm{Q}\|$ **then** add `(IMPLIES (Q ?X) (P ?X))` to *type-constraints*;
12. **return** *type-constraints*, *universal-types*

Figure 1: Algorithm for computing type-constraints. $Pred(p)$ is the predicate name of p.

- Incompatible types: {`(IMPLIES (Q ?X) (NOT (R ?X)))`}.

Theorem 1 *Find-type-constraints is sound and its complexity is* $O(|s_0| + |\Pi|^2|\Theta|)$ *(with the static predicates precomputed), where* Π *and* Θ *are respectively the set of type predicates and the set of constants in the planning domain.*

The soundness of Find-type-constraints is easily verified from the way argument sets of type predicates are extracted and compared; and $|s_0|$ in the complexity bound reflects steps 1-4, while the second term comes from the nested pair of *for*-loops. Of course, the sizes of Π and Θ are themselves bounded by the size of the initial state s_0, so that a looser bound is $O(|s_0|^3)$.

The number of type constraints actually found in various domains varies greatly, depending on the number of type predicates used in the domain models. For example, no type constraints may be found in blocks worlds, where the "block" property typically has no supertypes or subtypes, and the only object lacking the property is a unique table. By contrast, various transportation domains may yield dozens of type constraints. A few examples from a version of the "logistics" transportation world are

`(IMPLIES (AIRPORT ?X) (LOCATION ?X)), (IMPLIES (AIRPORT ?X) (NOT (CITY ?X))),`
`(IMPLIES (AIRPLANE ?X) (NOT (TRUCK ?X))),`

among a total of 15, found in a few milliseconds (on a low-end PC with a 800 MHz processor and only 256 Mbyte RAM). Other benchmark domains have more type constraints, such as Rovers for which 51 type constraints are still very quickly inferred. Since our purpose here is not to reiterate previously reported results, we refer the reader to [21, 16] for details.

3 The Hypothesize-and-Test Paradigm

We now turn to our hypothesize-and-test paradigm for finding a variety of constraints that can be guessed, and inductively verified, based on the precondition-effect structure

of the operators. In general, the number of syntactically possible state constraints in a planning domain is exponential in the number of literals allowed per formula; and, even if the number of literals is small (say 3 or 4), the number of combinations of signed predicates and constant or variable arguments can be very large – e.g., 10^8-10^{13} if there are several (e.g., 3) 2-place predicates and 10-20 constants.[5] Moreover, as we will elaborate, some constraints cannot be verified in isolation but only in combination with other constraints. Thus a pure enumerate-and-test approach seems unattractive, given that the goal is to find constraints quickly as a prelude to planning, or as feedback to planning domain developers. Instead, it is natural to seek guidance from the operators and initial conditions themselves in hypothesizing constraints.

There are various ways to do this, but as already illustrated in our overview, our particular approach was inspired by a rather striking observation: Most of the state constraints that intuitively characterize various planning domains, and that have played a role in the planning literature, are "hinted at" by pairs of effects, or by a single effect together with a persistent precondition (one that remains true when the operator is applied) in one of the operators. Recall, for example, that the blocks-world constraint that if `?X` is `ON ?Y` then `?Y` is not `CLEAR` is hinted at by the co-occurrence of effects `(ON ?x ?y)` and `(NOT (CLEAR ?y))` in the `Put` operator. Furthermore, since these effects depend on the precondition `(NEQ ?y TABLE)`, we also have an indication that this may be a required supplementary condition.

There is a good logical reason why we should expect valid state constraints to be "hinted at" by co-occurring effects (or effects and persistent preconditions): A state constraint holds in a given domain only if its truth is *maintained* by the operators. From a proof-theoretic point of view, we will be able to establish the truth of an implicative constraint only if we can show, by induction, that whenever an instance of the antecedent becomes true, the corresponding instance of the consequent becomes or remains true as well; and similarly for the contrapositive. In the blocks-world example at hand, we need an assurance that whenever the antecedent `(ON ?X ?Y)` becomes true for some values of `?X` and `?Y`, the consequent `(NOT (CLEAR ?Y))` becomes or remains true for those values (at least when `?Y` is not the `TABLE`), and similarly for the the contrapositive. Of course, this must be so not only in the operator that suggested the constraint, but in all operators, for all combinations of *when*-clauses whose preconditions might be simultaneously satisfied. Where these verification conditions fail, it may be possible to add supplementary conditions that are false whenever those failures occur, thus "rescuing" the constraint, albeit in a weakened form.

A possibility we have neglected in the preceding remarks is that operators might maintain the truth of a constraint implicitly through other constraints, rather than directly. For instance, it may be that whenever an operator has an effect matching the antecedent of a certain implication, the persistent preconditions of the operator entail the truth of the consequent *in virtue of other tacit state constraints*. Our strategy for discovering interdependent constraints is a pragmatic one. Our goal is not necessarily completeness, but rather to discover useful constraints (for planning and domain debugging) quickly. Thus we have examined a variety of domains to determine what sorts of interdependent constraints are common in practice, what inductive verification conditions suffice to confirm

[5] 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.

such combinations of constraints, and hence also how such combined constraints can be effectively hypothesized based on the precondition-effect structure of operators.

In particular, our observation has been that implicative constraints and sv-constraints are often interdependent, and so a major goal has been to detect and verify such interdependent constraints efficiently. The blocks-world example we have been using serves to illustrate this point as well. Consider the contrapositive of the previous constraint, stating that whenever `(CLEAR ?Y)` holds, `(NOT (ON ?X ?Y))` holds as well. Thus whenever an operator has an effect matching `(CLEAR ?Y)`, we need to be sure that `(NOT (ON ?X ?Y))` will also be true in the resultant state. Since all variables in a constraint are universally quantified, this means that no object `?X` must be `ON ?Y`. But blocks-world operators in the planning literature, like the `Put` operator given above, generally do not explicitly assert such a universally quantified effect; rather, they typically assert an effect `(NOT (ON ?x ?y))` for a *particular* `?x`, i.e., for an input parameter that will be bound to a constant in any instantiation of the operator. This is where the sv-constraint (indicated in context as `(ON ?*X ?Y)`) comes into play: If for a given value of `?Y`, there can be at most one value of `?X` for which `(ON ?X ?Y)` holds, then if an operator asserts effect `(NOT (ON ?x ?y))` while requiring `(ON ?x ?y)` as a precondition, we can be sure that in the resultant state `(ON ?X ?y)` no longer holds for *any* value of `?X`, i.e., nothing is `ON ?y`, as required. It turns out that the required sv-constraint also cannot be verified in isolation, i.e., it depends in turn on the truth of the implicative constraint.

This example also provides a clue as to when sv-constraints will be needed jointly with an implicative constraint we are trying to verify. The need for the sv-constraint arose from the fact that the antecedent of the implication, `(NOT (ON ?X ?Y))`, contains a variable (`?X`) not occurring in the consequent, `(CLEAR ?Y)`, where instances of the latter can be rendered false by some operator. (If `(CLEAR ?Y)` could not be rendered false – e.g., if `CLEAR` were a static predicate – then we would not have to verify the contrapositive!) This led to a universally quantified verification condition, which could only be indirectly confirmed via the sv-constraint `(ON ?*X ?Y)`. In general, then, we should expect to supplement an implicative constraint with one sv-constraint for each variable that occurs in one fluent literal but not in another fluent literal, where the latter can become false.

We have also developed other ways of expanding the range of constraints we can find, beyond those that can be guessed and verified in isolation using only the precondition-effect structure of operators. These ways include re-use of previously found constraints, and employing the hypotheses themselves (as opposed to verified constraints) to expand the preconditions of operators, allowing use of weaker verification conditions. These techniques are presented and discussed in [16].

3.1 Generic Algorithm

We now outline how we have implemented the above methodology, in a way that allows for multiple *when*-clauses in operators. We use several top-level programs to infer different types of constraints from operator structure, but most of them adhere to the following algorithmic structure. Note that Γ can be logically complex, for instance consisting of both an implicative hypothesis and one or two sv-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. In general, the idea is to choose the co-occurring literals and the corresponding hypothesis in such a way that the co-occurrence of the literals will (locally) support the inductive proof of the hypothesis.

(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.

In preparation for an illustration of the generic algorithm, we need to clarify the meaning of "persistent preconditions". We previously said that persistent preconditions in an operator are ones that remain true when the operator is applied. This will certainly be true for preconditions (whether primary or belonging to a *when*-clause) that are not contradicted by *any* effect of *any when*-clause of the operator (under the standard assumption that operator effects include all falsified preconditions). However, this is an unnecessarily strong requirement. Instead, our algorithms use the weaker notion of a *w-persistent* precondition, for any *when*-clause w of an operator, merely requiring the condition not to be unifiable with the negation of any effect of w or w_1 (the primary *when*-clause of the operator. (This is restated separately as Definition 1) below when we present our verification conditions formally.)

This raises the following question. Suppose that the ψ-part (consequent) of a hypothesized simple implicative constraint is based on a precondition that is w-persistent but not fully persistent, with respect to the operator o under consideration; i.e., operator o has some *when*-clause other than w with an effect that falsifies ψ. Is it still possible that the constraint holds? The answer is affirmative; in particular, the "offending" *when*-clause may have preconditions that never hold when the preconditions of w hold and the hypothesis being tested is true in the prior state; and even if the preconditions of the two *when*-clauses may hold simultaneously, we may be able to add supplementary conditions which are false whenever the preconditions of the "offending" *when*-clause are true.

An illustration

To help with the intuitive understanding of the generic algorithm, we illustrate it with an example from the blocks world, based on the `Put` operator shown in section 2. The illustration is necessarily informal, since we have not yet spelled out any details of steps (1-5), some of which depend on the particular type of constraint under consideration. But those details will be more understandable in light of the illustration to follow. The constraint whose discovery we outline is

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

This was also mentioned in section 2, except that we have now strengthened it by requiring `on` to be single-valued in its first argument, as indicated by the starred variable `?*X`.

(1) All *when*-clauses of all operators (here just one) are searched for potential implicative antecedents. For the (complex) type of constraint under consideration, these must be positive effects containing at least one parameter. One such potential antecedent, `(ON ?x ?y)`, is found in the final *when*-clause, which we will call w''. (Another possibility is `(CLEAR ?z)`, and both literals are also encountered in the preceding *when*-clause w', but we restrict our attention to the observations most relevant to the illustration.) An inner search loop then looks for a potential implicative consequent different from the potential antecedent. This is required to be a nonstatic literal (one whose predicate appears in some effect) meeting two conditions: (i) it is a persistent precondition or effect of w'' or of the "primary" *when*-clause w_1 (which in this case consists of the 5 preconditions `(ON ?x ?z), ..., (NEW ?x ?y)`, and no effects); and (ii) its parameters are a subset of those in `(ON ?x ?y)`. These requirements are met by the 3 preconditions of w_1 not involving `?z`, by both preconditions of w'', and by the last effect of w'', `(NOT (CLEAR ?y))`. We focus on this final option, so the corresponding implicative + sv-hypothesis is

```
(IMPLIES (ON ?*X ?Y) (NOT (CLEAR ?Y))).
```

Note that we have replaced operator parameters (lower case) by universal variables (upper case), and have "starred" variables of the antecedent not occurring in the consequent. (Without this additional assumption of single-valuedness, the inductive correctness proof would not go through.)

(2) We add candidate supplementary conditions to the constraint. These include the static preconditions of w_1 and w'' involving `?x` or `?y` only, i.e., `(NEQ ?x TABLE), (NEQ ?x ?y)`, and `(NEQ ?y TABLE)`. We also consider adding the negation of `(EQ ?y TABLE)`, a static precondition of the remaining *when*-clause w', but this is precluded because `(NEQ ?y TABLE)` unifies with (and in fact is the same as) a precondition of w''. So at this point the tentative constraint is

$\Gamma =$ `((IMPLIES (ON ?*X ?Y) (NOT (CLEAR ?Y)))`
`(NEQ ?X TABLE) (NEQ ?X ?Y) (NEQ ?Y TABLE)).`

(3) We look at all the effects of all the operators, checking whether any of them pose "threats" to the truth of Γ, assuming that it is true in the prior state. We use the verification conditions for this type of constraint (an implicative + sv-constraint, with subsumed variables in the consequent, discussed in detail later) to determine whether an apparent threat is real, and if so, which supplementary conditions could

be used to "excuse" the violation. Since the primary *when*-clause w_1 has no effects, it poses no threats to Γ. *When*-clause w', however, poses some threats. First of all, the effect `(ON ?x ?y)` threatens the sv-claim implicit in `(ON ?*X ?Y)`, since it could lead to multiple objects `?x` being placed on another object `?y`. At this point, we record the fact that the candidate supplementary condition `(NEQ ?Y TABLE)` can excuse any such violation, since it contradicts the precondition `(EQ ?y TABLE)` of w' (under the relevant binding `?x` $\leftrightarrow$ `?*X`, `?y` $\leftrightarrow$ `?Y`). The same effect `(ON ?x ?y)` also threatens the implication itself, since it might make the antecedent `(ON ?*X ?Y)` true without at the same time making `(NOT (CLEAR ?Y))` true (since the latter is not an effect of w'). Again, the supplementary condition `(NEQ ?Y TABLE)` is identified as a possible excuse for the apparent violation.

The effect `(CLEAR ?z)` of w' also appears to threaten Γ, since it contradicts the consequent `(NOT (CLEAR ?Y))` under the binding `?z` $\leftrightarrow$ `?Y`). But this threat turns out not to be real, since the primary precondition `(ON ?x ?z)` and w'-effect `(NOT (ON ?x ?z))` together with the assumption that Γ holds in the prior state ensure that `(ON ?*X ?z)` is false for all possible values of `?*X` (nothing is on `?z`) in the resultant state, preserving the truth of the implication. (We leave the details of this argument to the proof of correctness of the verification conditions.)

Similarly *when*-clause w'' appears to pose threats to Γ through its effects `(ON ?x ?y)` and `(CLEAR ?z)`. However, none of the threats are real. The apparent threat by `(ON ?x ?y)` to the sv-condition in `(ON ?*X ?Y)` is not real because the w''-precondition `(CLEAR ?y)` ensures, via the contrapositive of Γ, that `?x` will be the *only* object on `?y`. The apparent threat by `(ON ?x ?y)` to the implication is not real because of the simultaneous effect `(NOT (CLEAR ?y))`, which assures truth of the consequent (this of course was the basis for postulating Γ in the first place). And the apparent threat by `(CLEAR ?z)` to the implication is not real for the same reason as in w'. So since `(NEQ ?Y TABLE)` was the only candidate supplementary condition that was invoked as a potential excuse for apparent violations of Γ, the hypothesized constraint is now

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

(4) Finding minimal subsets of the supplementary conditions that cover all apparent violations of Γ is trivial in this example. The only such subset is obviously the singleton set containing `(NEQ ?Y TABLE)`. So Γ remains as above.

(5) We now verify the truth of Γ in the initial state, by systematically looking for counterexamples. A counterexample must have `(ON` C_1 C_2`)` and `(CLEAR` C_2`)` true, where $C_2 \neq$ `TABLE`, for some pair of constants C_1, C_2 occurring in the initial state. So we find all instances of form `(ON` C_1 C_2`)` where $C_2 \neq$ `TABLE`, and if a corresponding literal `(CLEAR` C_2`)` is present as well, we have a counterexample. If there are no such counterexamples, Γ is verified.

3.2 Finding Verification Conditions

As mentioned in our initial overview, formulation of verification conditions and of procedures that efficiently test them remains something of an art, in the absence of general practical proof procedures for mathematical induction, applicable to dynamic worlds of the sort considered in automated planning.

However, the intuitive idea behind the formulation of verification conditions is quite straightforward. Setting aside sv-conditions for the moment, imagine a constraint written as a disjunctive clause $\phi_1 \vee ... \vee \phi_n$, where the ϕ_i are positive or negative literals containing constants or variables. Whenever a substitution instance $(\phi_i)_u$ of a literal ϕ_i of the clause is rendered false by the effects of an operator, we want to ensure that the clause as a whole remains true (for all values of its variables), assuming that it was true (for all values of its variables) in the state in which the operator was applied. For this it is sufficient (though not necessary) that there be some other literal ϕ_j, such that $(\phi_j)_u$ will be true for all values of its variables (if any are not bound by substitution u), in the resultant state. If $(\phi_j)_u$ has no unbound variables (i.e., the variables of ϕ_j are a subset of those of ϕ_i), then it is sufficient that $(\phi_j)_u$ be a persistent precondition of the operator instance in question, or that it be an effect of that operator instance. If $(\phi_j)_u$ has unbound variables, then its truth for all values of those variables may depend on additional assumptions about the state space, in particular on sv-constraints; we postpone further details of this case. In any event, we are led in this way to a set of verification conditions, separately motivated by the possibility of each literal ϕ_i becoming false.

In the case of (added or stand-alone) sv-constraints, our verification conditions must guard against both explicit and implicit violations of single-valuedness. An explicit violation of an sv-constraint `(P ?*X ?Y)` would occur if an operator instance could have two effects matching `(P ?X ?Y)`, with different bindings for `?X` and the same binding for `?Y` (and similarly when there are additional "starred" and "unstarred" variables). So our verification conditions must rule out such concurrent effects. But even in the absence of such explicit violations, we need to guard against implicit violations whenever an operator has an effect matching `(P ?X ?Y)`. For if the operator has an effect `(P A B)` when applied in a state where `(P A' B)` already holds, with `A'` distinct from `A`, then `P` is multi-valued in its first argument for second argument `B`. To be sure that this cannot occur, we therefore need to verify either that (a) no instance of `(P ?X B)` could have been true in the prior state, or that (b) there is a "compensating" change, i.e., the effect `(P A B)` will be offset by a *change* from a precondition `(P A' B)` to a resultant condition (effect) `(NOT (P A' B))`. Condition (a) is the more problematic one, since it cannot be verified independently of additional knowledge about state constraints. Thus, just as the verification of certain clausal constraints depends on simultaneous verification of certain sv-constraints (as noted above), the converse is also true. Thus there are clausal (implicative) and sv-constraints that can only be verified jointly.

This discussion indicates that there are just a few distinct *types* of verification conditions that will be needed to deal with a rather broad range of (separate or simultaneous) clausal and sv-constraints. We can enumerate these as follows.

(i) *Forbidden multiple effects*: we 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) *Concomitant effects*: we 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) *Concomitant changes*: we 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);

(iv) *Compensating changes*: we require that a positive effect (asserting an instance of a single-valued predicate) be compensated by a change from a positive instance to a negative instance of the same predicate; (for several types of sv-hypotheses posited in isolation or in combination with an implicative hypothesis, this check can be accomplished just by looking for appropriate concomitant changes; however, for testing single-valuedness of the predicate in the antecedent of an implicative hypothesis, where the variables of that predicate subsume those in the consequent, we can use a more subtle method that avoids checking for a compensating change whenever the assumed truth of the implicative hypothesis prior to application of an operator makes this unnecessary;) and

(v) *Unwanted concurrent effects*: we preclude the co-occurrence of certain effects with certain other effects (e.g., this is needed in testing exclusion hypotheses, stating that the truth of one predication implies the falsity of another).

3.3 Collecting 'Excuses'

Procedurally, the most complex aspect of the hypothesize-and-test paradigm in section 3.1 is the collection of possible "excuses" (sets of candidate supplementary conditions) in step 3, when verification conditions for Γ are violated. Our programs for testing hypotheses and collecting "excuses" are organized around 5 subroutines corresponding to the 5 types of verification conditions we have enumerated.

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 different). 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 5 basic routines for collecting excuses for violations of hypothesized constraints, corresponding to the 5 types of verification conditions, are the following: (1) `test-sv-effect`, (2) `test-concomitant-effect`, (3) `test-concomitant-change`, (4) `test-compensating-change`, and (5) `test-unwanted-effect`. We will allude to some of these routines in our discussion of verification conditions for various types of constraints, but leave further details to a technical report [16].

3.4 Finding Minimal Sets of Supplementary Conditions

As indicated in the 4th step of the generic hypothesize-and-test algorithm, once "excuses" have been collected for all apparent failures of a hypothesis, we try to select irredundant sets of supplementary conditions, up to a certain size, where each such set suffices to excuse all apparent failures. Since there may be multiple alternative excuses for any given failure, the goal is to ensure that the set of supplementary conditions covers the candidate supplementary conditions of at least one excuse for each failure. (Recall that in cases where an excuse serves to ensure that a required effect provably occurs, the excuse may contain multiple candidate supplementary conditions.)

Since this problem subsumes the set covering problem (e.g., [11]),[6] it is NP-hard in terms of the sum, over the candidate supplementary conditions, of the number of violations that a candidate supplementary condition can excuse. For this reason we originally used a polynomial-time greedy algorithm for finding a small (not necessarily minimal) set of supplementary conditions. While this worked well, giving minimal sets in all cases we tried, the fact that in principle there can be multiple minimal sets – and each of these leads to a distinct state constraint – motivated us to switch to a complete algorithm, delivering *all* minimal sets up to a certain size. In all domains we have tested so far, no more than two supplementary conditions were ever needed (no new constraints were found if up to 5 supplementary conditions were allowed); consequently, the algorithm for finding minimal sets of supplementary conditions, though exponential in principle, has not been a major factor in the overall time complexity of DISCOPLAN. The complete algorithm and relative complexity analysis are detailed in [16].

3.5 Filtering Constraints through the Initial State

Each possible state constraint, including an irredundant set of supplementary conditions, is tested in step 5 of the generic hypothesize-and-test algorithm against the initial conditions. This is done by attempting to find a counterexample, as outlined for the case of implicative constraints in Figure 2. Before the test is applied, all `EQ/NEQ`-literals other than unsigned `NEQ`-literals in the antecedent are eliminated. The elimination of other `EQ/NEQ`-literals can be understood by imagining all such literals to be first moved into the antecedent of an implicative constraint (with a change in sign if coming from the consequent). Suppose that negated `NEQ`-literals are rewritten in terms of `EQ`, and *vice versa*. Then antecedent literals of form `(EQ` c c`)` or `(NEQ` c c'`)` are clearly redundant and can be dropped (for distinct constants c, c'); ones of form `(EQ` c c'`)` or `(NEQ` c c`)` render the constraint trivially true and thus of no further interest; and ones of form `(EQ` u v`)` where at least one of u, v is a variable can be dropped after substituting the second (possibly nonvariable) term for the variable term in all other literals. This leaves us with unsigned `NEQ`-literals only. These could be transferred as disjoined `EQ`-literals to the consequent.

The algorithm rests on the following observations. Consider an implicative constraint of form $((\phi_1 \wedge ... \wedge \phi_m) \Rightarrow (\psi_1 \vee ... \vee \psi_n))$, where all literals ϕ_i and ψ_j (including, possibly, `EQ`-

[6] Essentially, finding a minimal set of supplementary conditions, for the case where excuses are singleton sets of supplementary conditions, is the dual of the set covering problem. More precisely, the translation to the set covering problem is obtained by interpreting the relation "$\{x\}$ is one of the excuses in the set of alternative excuses y (for a particular violation)" as "$y \in x$" (where x is one of the sets available as part of the covering we are seeking). The equivalence is most easily seen by drawing the x-elements and y-elements as vertices of a graph, connected by edges corresponding to the stated binary relations.

Algorithm: *Initially-refute-state-constraint*(*SC*, *inits*)

SC: an implicative state constraint `((IMPLIES` ϕ ψ`)` $\sigma_1 ... \sigma_k$`)`; the σ_i are generally static literals (though this is not required);

inits: a list of initial conditions (with absence interpreted as negation);

Output: If the constraint is refuted, a counterexample `((and` ϕ_u $\neg\psi_u$`)` $(\sigma_1)_u ... (\sigma_k)_u$`)` is given, for some ground substitution u for the variables of ϕ. If the constraint is verified, the output is `nil`.

1. Let $\phi_1, ...\phi_m$ be the positive literals among $\phi, \overline{\psi}, \sigma_1, ..., \sigma_k$;
2. Let $\psi_1, ..., \psi_n$ be the complements of the negative literals among $\phi, \overline{\psi}, \sigma_1, ..., \sigma_k$;
   /* We assume there may be `EQ`-literals among the ψ_j, while other `EQ/NEQ`-literals have been eliminated; */
3. Let U be the list of all unifiers (bindings of all variables occurring in $\phi_1, ..., \phi_m$ to constants occurring in *inits*) such that for $u \in U$, $\{(\phi_1)_u, ..., (\phi_m)_u\}$ is a subset of *inits*;
   /* if $m = 0$, $U =$ `(T)`, where `T` is the trivial unifier; and if there are no unifiers, $U =$ `nil`;*/
4. **if** $U =$ `nil` **then return** `nil` /* no counterexample exists */;
5. **for** each u in U **do**
   (a) Let v be a "missing binding" for $(\psi_1)_u, ..., (\psi_n)_u$ relative to *inits*, if one exists; i.e., v binds the variables occurring in $(\psi_1)_u, ..., (\psi_n)_u$ (if any) to constants occurring in *inits* in such a way that *none* of $(\psi_1)_{uv}, ..., (\psi_n)_{uv}$ occur in *inits* or are of form `(EQ` c c`)`;
   {if $n = 0$, $v =$ `T`, and if there is no "missing binding", $v =$ `nil`;}
   (b) **if** $v \neq$ `nil` **then return** `((and` ϕ_{uv} $\neg\psi_{uv}$`)` $(\sigma_1)_{uv} ... (\sigma_k)_{uv}$`)`
   else continue with next binding u in U;
6. /* No counterexample exists - The implicative constraint has been verified */
   return `nil`.

Figure 2: Algorithm for refuting an implicative constraint in the initial state.

literals among the ψ_j) are unnegated and all variables are treated as wide-scope universals. Any constraints we consider can be written in this form. To verify the constraint in the initial conditions, we must be able to verify that for every (simultaneous) substitution instance of $\{\phi_1, ..., \phi_m\}$ that occurs in the initial conditions, $(\psi_1 \vee ... \vee \psi_n)$ is also verifiable in the initial conditions, *for all values of its remaining unbound variables*. So the algorithm first finds all unifiers (bindings of variables to constants) u such that $\{(\phi_1)_u, ..., (\phi_m)_u\}$ is a subset of the initial conditions. For each unifier u it then tries to verify $((\psi_1)_u \vee ... \vee (\psi_n)_u)$ for all possible bindings of the remaining variables (if any), returning a counterexample if it finds one. If $((\psi_1)_u \vee ... \vee (\psi_n)_u)$ has no unbound variables (i.e., the variables in the ϕ_i include those in the ψ_j), then it is verified if one of the disjuncts is a literal `(EQ` c c`)`, or occurs in the initial conditions. If u does not bind all variables in the ψ_j, then we need to verify that for every way of binding the remaining variables to constants occurring in the initial conditions, one of the disjuncts is a literal `(EQ` c c`)` or occurs in the initial conditions. If for some binding none of the disjuncts have this property, we have a counterexample.

In step (3) of the algorithm, the complexity of the search for all bindings of the ϕ_i is $O(l_c^2 n_{init})$, where l_c is the size (number of symbols) of the hypothetical constraint and n_{init} is the number of initial conditions. If the ϕ_i contain all variables occurring in the ψ_j, then the "missing binding" search in step (5a) is just a check whether some $(\psi_j)_u$ is of form `(EQ`

c c) or occurs in *inits*, which is $O(l_c n_{init})$ (if we simply use a list for *inits*). If there are additional variables in the $(\psi_j)_u$, then the "missing binding" search is $O(l_c n_{init}^{n_{var}})$, where n_{var} is the maximum number of variables any one of the $(\psi_j)_u$ can have (i.e., after binding of all variables occurring in the ϕ_i). Thus the overall complexity is $O(l_c n_{init}^{n_{var}} + l_c^2 n_{init})$ or $O(l_c^2 n_{init})$, depending on whether or not the ψ_j contain variables occurring in the ϕ_i.

There is also an algorithm called *Initially-refute-sv-constraint* that carries out an analogous refutation attempt for an sv-hypothesis with supplementary conditions, written as $(\phi^*\ \sigma_1 ... \sigma_k)$, where ϕ^* is a positive predication containing *-variables, indicating that it is true for at most one tuple of values of those variables for any given values of the remaining variables, whenever the supplementary conditions $\sigma_1 ... \sigma_k$ (with the corresponding substitutions) hold. The algorithm is straightforward and we will omit details. In essence, the refutation attempt consists of (1) finding all unifiers of ϕ^* with predications in the initial conditions, (2) filtering out unifiers for which the corresponding instances of $\sigma_1 ... \sigma_k$ do not hold in the initial conditions, and (3) for the remaining unifiers, searching for pairs that assign distinct values to some *-variables while assigning the same values to other variables. The complexity of the algorithm, as we have implemented it, is $O(l_c n_{init}^2)$, where l_c is again the length of the sv-constraint being tested. This could be improved, to secure maximal efficiency for large sets of initial conditions, to $O(l_c n_{init})$ on average, by suitable hash-coding of formulas (using predicates and non-starred argument positions as keys).

For concreteness, we conclude this section with a few examples of constraints found by the above methods for the "logistics" transportation domain (with supplementary conditions incorporated into the antecedent of implications):

```
(IMPLIES (AND (AT ?X ?Y) (AIRPLANE ?Y)) (AIRPORT ?Y))
(IMPLIES (AND (AT ?X ?Y) (OBJECT ?Y)) (NOT (IN ?X ?Y)))
(IMPLIES (AND (AT ?X ?Y) (AT ?X ?V0) (TRUCK ?Y)) (EQ ?Y ?V0)).
```

These are respectively an implicative constraint, an exclusive state constraint (previously mentioned), and an sv-constraint. A total of 30 constraints (including the 15 type constraints mentioned earlier) were obtained in this version of the logistics domain in 56 milliseconds. As before, more specifics can be found in [21, 16].

3.6 Remarks on the Utility of Bootstrapping in Hypothesis Testing

In testing new hypotheses, it would seem natural to augment operator preconditions with (a) consequences of the constraints already verified, and (b) consequences of the hypotheses being tested. Note that (b) is legitimate since we are verifying whether operators maintain the truth of constraints *given* that the constraints hold in the antecedent state; and such augmentation could improve our chances of successfully confirming constraints, because the added consequences may provide persistent preconditions needed for the verification.

In [16], we provide a careful discussion of the trade-offs in this kind of bootstrapping, and describe the bootstrapping methods implemented in DISCOPLAN. Concerning (a), most of the constraints of interest that depend on the truth of other constraints are implicative constraints dependent on sv-constraints, and our algorithms for such combinations find these quite effectively. Concerning (b), if consequences of yet-to-be-verified hypotheses are added to operator preconditions, many different expanded versions of those preconditions may need to be tried, corresponding to various ways of "rescuing" a hypothetical invariant by adding supplementary conditions. Also, we show that isolated testing of a hypothesis, using just its own consequences to augment operator preconditions, is unlikely to be helpful,

except in certain special cases. (The "walking and taking a cab" example in section 4.3.1 is such a case.)

However, we have found three bootstrapping methods to be somewhat useful: The first of these methods is aimed specifically at antisymmetry constraints. For these we expand operator preconditions using invariants found in the first stage, and also expand effects in a way that is designed to lead to contradiction if the antisymmetry hypothesis is false. The second bootstrap method simply runs the basic discovery techniques iteratively, adding consequences of previously verified constraints to operator preconditions at each cycle. The third method applies the basic verification methods "in parallel" to (small) sets of hypotheses that have not yet been verified, always using operators whose preconditions have been augmented with the consequences of those same hypotheses, as well as of previously confirmed hypotheses. These methods are close in spirit to those of Rintanen [36]. (The main difference is our very selective way of formulating hypotheses, and collecting and choosing sets of static supplementary conditions).

Our basic methods yield the majority of constraints and do so very quickly. The bootstrap methods are more time-consuming, but occasionally yield additional constraints. This paper focuses on the basic methods, which are powerful enough in practice to discover the majority of state constraints that the full version of DISCOPLAN infers for a large class of benchmark domains [16].

4 Basic Hypothesize-and-Test Techniques

In the next three subsections, we describe our basic instantiations of the hypothesize-and-test paradigm. The first two subsections consider implicative and sv-constraints in isolation. While many constraints can be found in this way, many others require combined consideration of implicative and sv-constraints, as described in the third subsection.

4.1 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. In keeping with our earlier remarks, 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 (as defined in section 3.1), 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.

We ensure that these requirements are met in step (1) of the hypothesize-and-test paradigm by choosing ϕ to correspond to some effect of some *when*-clause w of some operator o (so that ϕ will be a fluent literal); and choosing ψ to correspond to an effect or persistent precondition of *when*-clause w or of the primary *when*-clause w_1 of o (if distinct), where ϕ contains all the variables occurring in ψ, and if ϕ contains variables not

occurring in ψ, then ψ is upward monotonic. We make one exception to the above method of hypothesizing simple implicative constraints, in the interest of minimizing unproductive effort: When the chosen effect ϕ is a negative literal and ψ is a static literal (chosen from the preconditions of o) whose variables are a proper subset of those of ψ, we do not further pursue the corresponding hypothesis $((\phi \Rightarrow \psi)...)$. This is based on the empirical observation that hypotheses that posit a static constraint on a subset of the variables of a negated fluent predicate are far less likely to hold than for the unnegated predicate. (For example, blocks-world operators may suggest both `(IMPLIES (NOT (ON ?X ?Y)) (BLOCK ?X))` and `(IMPLIES (ON ?X ?Y) (BLOCK ?X))` as simple implicative hypotheses, but unless everything, including the table, is considered a block, only the latter will turn out to hold.) Note that if the above method of forming simple implicative hypotheses is applied systematically to all effects (as ϕ-candidates) in combination with all eligible preconditions and effects (as ψ-candidates), then if ϕ and ψ both correspond to effects and contain the same sets of variables, both $((\phi \Rightarrow \psi)...)$ and its converse $((\psi \Rightarrow \phi)...)$ will be tried as simple implicative hypotheses. (The dots indicate that supplementary conditions are indeterminate at this point in the hypothesis formation process.)

As anticipated in the discussion in section 4.2, the above syntactic constraints on ϕ and ψ, and the way ϕ and ψ are chosen, are motivated by the verification conditions for an implicative constraint. Recall in particular that if the constraint is to be verified independently of any sv-constraints, the consequent ψ should either contain all the variables occurring in ϕ (so that we will not require a simultaneous sv-constraint on ϕ), or should never be falsified by an operator effect (so that the truth of the contrapositive of the implication is trivially maintained). It is in virtue of their independent verifiability that such constraints are termed "simple".

Apart from notational preliminaries, we are now ready to state the verification conditions of a simple implicative constraint formally, and to prove that these conditions guarantee the truth of the constraint. First we formally restate the definition of w-persistent preconditions, where the symbol w_1 denotes the primary *when*-clause corresponding to a *when*-clause w of an operator o (where $w_1 = w$ if w is itself primary), and *preconds*(w) denotes the set of preconditions (including `EQ/NEQ`-conditions) of *when*-clause w:

Definition 1 (w-persistent condition) *A precondition ψ belonging to* preconds(w) *of operator o is w-persistent if and only if $\neg\psi$ is not unifiable with any effect of w or w_1.*

We will also write *preconds*$^+(w)$ for the combined preconditions of *when*-clauses w and w_1. Also $\overline{\phi}$ stands for the complement (negation) of a literal ϕ. *Unification* (or informally, *matching*) of two literals ϕ and ϕ' is in general constrained by the `EQ/NEQ`-preconditions of a particular *when*-clause w and the primary *when*-clause w_1 of a particular operator o. Variables occurring in ϕ and ϕ' may either be "external" universal variables (belonging to a hypothesis that we are testing), or parameters of operator o. Successful unification requires that the literals have the same polarity and predicate, and allows for (i) trivial unification (with unifier T) of identical terms and of terms belonging to the same `EQ`-set of w in combination with w_1 (where these sets are precomputed in standardized operators); (ii) substitution of a constant, parameter, or variable for a variable; (iii) substitution of a constant or parameter κ for a parameter α, provided that w and w_1 do not have a `NEQ`-precondition relating α to κ, or relating α to a member of the `EQ`-set of κ. Note that substitution of variables for parameters is precluded.

In the verification conditions that follow (as well as in the verification conditions for other types of constraints), we will use equivalence symbols, entailment symbols, and equality signs subscripted with the names of one or more *when*-clauses, in statements such as $\psi'_u \equiv_w \psi_u$, $preconds^+(w_2)_{uu'} \models_w \neg(\sigma_i)_{uu'}$, and $\underline{X}_u =_{w,w_2} \underline{X}_v$. The subscripted name of a *when*-clause indicates that the `EQ/NEQ`-preconditions of the *when*-clause, *along with those of the primary when*-clause w_1, may be used in establishing the equivalence, consequence relation, or equality in question. However, the intention is that these `EQ/NEQ`-preconditions first be particularized using the substitutions appearing as subscripts on both sides of the relation. This slightly loose notation (avoiding additional subscripting) should cause no confusion and will be used often hereafter. As an illustration of some theoretically possible subtleties in the process of unifying literals of a hypothesis with literals occurring in an operator, and subsequently establishing an equivalence, consequence or equality relation, suppose that we unify a hypothesis literal `(P ?X ?X ?Y)` with an effect literal `(P a ?x ?y)` of some *when*-clause w of some operator o. Thus the unifier will be

$u =$ `(a/?X)(a/?x)(?y/?Y)`,

assuming that there are no `EQ/NEQ`-preconditions in w that affect this unification. Suppose that subsequently we unify the same hypothesis literal `(P ?X ?X ?Y)` with a precondition `(P ?r ?s ?z)` of w, obtaining unifier

$v =$ `(?r/?X)(?r/?s)(?z/?Y)`,

again assuming no interference from `EQ/NEQ`-preconditions. (Such a second unification would be performed if, for instance, we were trying to establish that `P` is single-valued in its third argument whenever the first two arguments are the same – see the "dedicated" method for discovering sv-constraints in the next subsection.) Now suppose that the only `EQ`-precondition in w is

`(EQ ?x ?s)`,

and the verification conditions we are checking require that the two matches of the hypothesis literal `(P ?X ?X ?Y)` against the two operator literals should yield identical values for the first two arguments, given the relevant `EQ`-preconditions; i.e.,

$\langle$`?X,?X`$\rangle_u =_w \langle$`?X,?X`$\rangle_v$.

Then this is indeed true, but this becomes clear only when we take proper account of the assumed `EQ`-precondition. Without that precondition the equation is

$\langle$`a,a`$\rangle = \langle$`?r,?r`$\rangle$.

Even the `EQ`-precondition `(EQ ?x ?s)` taken at face value does not immediately confirm the desired identity. Only when we particularize the `EQ`-literal to

`(EQ ?x ?s)`$_{uv}$ = `(EQ a ?r)`

does the correctness of the identity come to light. Another point we should mention is that equivalences and identities with one or more subscripted *when*-clause names may also appear in *negated* form. Their meaning should be clear by analogy with the unnegated case. For example, `?Y`$_u \neq_w$ `?Y`$_v$ means that the left-hand side cannot be identical with the right-hand side, given the `EQ/NEQ`-preconditions of w, where these have been particularized using substitutions u and v. (In particular, this would be true for the above values of u and v, if w had a precondition `(NEQ ?y ?z)`.)

As a final preliminary, we emphasize that the equivalence and entailment relations we employ could be interpreted in a way that makes use not only of the `EQ/NEQ`-preconditions of the indicated *when*-clauses, but of other relevant knowledge, such as type constraints. In fact, our implementation of contradiction tests between operator preconditions and a supplementary condition of a hypothesis, such as

$preconds^+(w_2)_{uu'} \models_w \neg(\sigma_i)_{uu'}$,

makes uniform use of a '`contradicts`' function that employs not only `EQ/NEQ`-preconditions but also type constraints to detect a contradiction. In particular, it detects not only whether one of two predications π, σ is the negation of the other, but also, when both are monadic, whether the two predicates are known to be incompatible, or whether one is the negation of a known supertype of the other.

Definition 2 (Verification conditions for a simple implicative constraint) *Let* $\Gamma = ((\phi \Rightarrow \psi)\ \sigma_1 ... \sigma_k)$. DISCOPLAN*'s verification conditions for* Γ *are:*

(1) Γ *is true initially, for all substitutions of constants in the planning domain (i.e., occurring in the initial conditions – recall footnote 4) for variables in* Γ.

(2) If a when*-clause* w *of an operator* o *has an effect* ϕ' *matching* ϕ *with unifier* u*, then one of the following holds:*

- *(a) (Concomitant effect, same when-clause)* w *or* w_1 *has an effect* ψ' *such that* $\psi'_u \equiv_w \psi_u$*; or*
- *(b) (Persistent precondition)* w *or* w_1 *has a* w*-persistent precondition* ψ' *such that* $\psi'_u \equiv_w \psi_u$*, and for every* when*-clause* w_2 $(\neq w, w_1)$*, if* w_2 *has an effect* $\overline{\psi}''$ *matching* $\overline{\psi}_u$ *with unifier* u'*, then* $\text{preconds}^+(w_2)_{uu'} \models_w \neg(\sigma_i)_{uu'}$*, for some* i $(1 \leq i \leq k)$*; or*
- *(c) (Concomitant effect, different when-clause)* o *has a* when*-clause* w_2 $(\neq w, w_1)$ *such that* $(\sigma_1)_u, ..., (\sigma_k)_u \models_w \text{preconds}(w_2)_u$ *and* w_2 *has an effect* ψ' *such that* $\psi'_u \equiv_{w,w_2} \psi_u$*; or*
- *(d) (False supplementary condition)* $\text{preconds}^+(w)_u \models \neg(\sigma_i)_u$*, for some* i $(1 \leq i \leq k)$.

(3) If a when*-clause* w *of an operator* o *has an effect* $\overline{\psi}'$ *matching* $\overline{\psi}$ *with unifier* u*, then one of the following holds (these alternatives correspond closely to (a-d) above):*[7]

- *(a)* w *or* w_1 *has an effect* $\overline{\phi}'$ *such that* $\overline{\phi}'_u \equiv_w \overline{\phi}_u$*; or*
- *(b)* w *or* w_1 *has a* w*-persistent precondition* $\overline{\phi}'$ *such that* $\overline{\phi}'_u \equiv_w \overline{\phi}_u$*, and for every* when*-clause* w_2 $(\neq w, w_1)$*, if* w_2 *has an effect* ϕ'' *matching* ϕ_u *with unifier* u'*, then* $\text{preconds}^+(w_2)_{uu'} \models_w \neg(\sigma_i)_{uu'}$*, for some* i $(1 \leq i \leq k)$*; or*
- *(c)* o *has a* when*-clause* w_2 $(\neq w, w_1)$ *such that* $(\sigma_1)_u, ..., (\sigma_k)_u \models_w \text{preconds}(w_2)_u$ *and* w_2 *has an effect* $\overline{\phi}'$ *such that* $\overline{\phi}'_u \equiv_{w,w_2} \overline{\phi}_u$*; or*
- *(d)* $\text{preconds}^+(w)_u \models \neg(\sigma_i)_u$*, for some* i $(1 \leq i \leq k)$.

Example. In the blocks-world formalization with the `Put` operator introduced in section 1.2, the hypothesis Γ = `((IMPLIES (ON ?X ?Y) (NEQ ?X TABLE)))` satisfies verification condition (2.b) of Definition 2. Moreover, condition (3) of Definition 2 is also satisfied, because `Put` has no *when*-clause matching `(NOT (NEQ ?X TABLE))` (or `(EQ ?X TABLE)`). Thus, if Γ is true initially, i.e., condition (1) of Definition 2 holds, then Γ is a valid state constraint.

[7] Given the definition of simple implicative constraints, the variables of ψ must be the same as those of ϕ at this point. (However, since some arguments may be constants, ϕ and ψ need not have the same sets of arguments.)

Our procedures do not directly implement the stated verification conditions. Recall that we test hypotheses $((\phi \Rightarrow \psi)\ \sigma_1...\sigma_n)$ wherein $\sigma_1...\sigma_n$ are *candidate* supplementary conditions, from which minimally adequate subsets are to be selected (see steps (2-5) of the hypothesize-and-test algorithm). So in practice, we search for cases where verification conditions (2) or (3) are relevant, and identify two types of apparent failures: cases where there is neither an accompanying effect as specified in (a) nor a w-persistent precondition as specified in (b); and cases where there is a w-persistent precondition as specified in (b), but this is potentially subverted by an additional effect in another *when*-clause negating the w-persistent precondition. We then identify possible alternative "excuses" for all such failures. For failures of the first type, excuses are singleton supplementary conditions as specified in (d) (making the effect ϕ' specified in (2) or the effect $\overline{\psi}'$ specified in (3) irrelevant), or sets of supplementary conditions as specified in (c) (supplying the desired effect accompanying ϕ' or $\overline{\psi}'$ via another *when*-clause). For failures of the second type, an excuse can again be a singleton as per (d), or a singleton as per the second part of (b) (making the "subversive effect" irrelevant), or a set of supplementary conditions as per (c) (ensuring through an effect of another *when*-clause that the desired condition persists after all, despite its apparent subversion in (b)).

The proofs of the following theorems are given in Appendix A.

Theorem 2 *If a simple implicative constraint* Γ *meets the verification conditions of Definition 2 in a planning domain, then it holds in all states reachable from the initial state. (Hence simple implicative constraints found by* DISCOPLAN *are correct.)*

Theorem 3 *The computation of simple implicative constraints in accordance with the hypothesize-and-test scheme is completed in polynomial time for any bound* max *on the allowable number of supplementary conditions, specifically in*

$$O\left(\sum_{o\in O}\left[l_o^3\left(\sum_{o'\in O} l_{o'}^2 + \frac{max}{l_o}\cdot\sum_{o'\in O} l_{o'}^{max+2} + l_o n_{init}\right)\right]\right)$$

steps, where O *is the set of planning operators,* l_o *is the size of operator* o*, and* n_{init} *is the number of initial conditions.*

4.2 Single-Valuedness Constraints: "Dedicated" Search

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` 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 change from a P-precondition to a corresponding $\neg P$-effect that appears to maintains single-valuedness. The verification conditions ensure that there are no multiple effects that could violate single-valuedness (see clause (2)), and that for any P-effect there is a concomitant change capable of keeping P single-valued (see clause (3)).

Definition 3 (Verification conditions for an sv-constraint) *Let* $\Gamma = (\phi^*\ \sigma_1 ... \sigma_k)$ *be an sv-constraint where* ϕ^* *is a positive literal containing 0 or more "unstarred" variables* $\underline{X}$ *and 1 or more "starred" variables* $*\underline{Y}$ *(and possibly constants), and* σ_1, ..., σ_k *may contain variables of* $\underline{X}$ *and* $*\underline{Y}$. DISCOPLAN*'s verification conditions for* Γ *are:*

(1) The sv-constraint is true initially, for all substitutions of constants in the planning domain for variables of the constraint.

(2) (No simultaneous multiple effects) If an operator o has when*-clauses w and* w_2 *(possibly identical) such that w has an effect* ϕ *matching* ϕ^* *with unifier u, and* w_2 *has another effect* ϕ' *such that* ϕ'_u *matches* ϕ^* *with unifier v, where we cannot disprove* $\underline{X}_u = \underline{X}_v$ *(i.e., their constrained unifier exists, given the* `EQ/NEQ`*-preconditions of w,* w_1 *and* w_2*) and cannot prove* $*\underline{Y}_u = *\underline{Y}_v$ *(i.e., they are not identical even when we appeal to the* `EQ`*-preconditions of w,* w_1 *and* w_2*), then one of the following must hold:*

(a) $\text{preconds}^+(w)_u \models \neg(\sigma_i)_u$ *for some i* $(1 \leq i \leq k)$*; or*

(b) $\text{preconds}^+(w_2)_v \models \neg(\sigma_i)_v$ *for some* $(1 \leq i \leq k)$*.*

(3) (No cumulative multiple effects) If an operator o has a when*-clause w such that w has an effect* ϕ *matching* ϕ^* *with unifier u, then one of the following must hold:*

(a) (Concomitant change) w or w_1 *has a precondition* ϕ' *where* ϕ'_u *matches* ϕ^* *with unifier v such that* $\underline{X}_v =_w \underline{X}_u$ *and* $*\underline{Y}_v \neq_w *\underline{Y}_u$;[8] *and either*

i. w or w_1 *has an effect* $\overline{\phi}''$ *where* $\overline{\phi}''_{uv}$ *matches* $\overline{\phi}^*$ *with unifier* v' *such that* $\underline{X}_v =_w \underline{X}_{v'}$ *and* $*\underline{Y}_v =_w *\underline{Y}_{v'}$*; or*

ii. some secondary when*-clause* w_2 *other than w has an effect* $\overline{\phi}''$ *where* $\overline{\phi}''_{uv}$ *matches* $\overline{\phi}^*$ *with unifier* v' *such that* $\underline{X}_v =_{w,w_2} \underline{X}_{v'}$ *and* $*\underline{Y}_v =_{w,w_2} *\underline{Y}_{v'}$ *and furthermore* w_2 *has static preconditions that are jointly entailed by the supplementary conditions, i.e.,* $(\sigma_1)_{v'}, ..., (\sigma_k)_{v'} \models_w \text{preconds}(w_2)_{v'}$*; or*

(b) (False supplementary conditions) $\text{preconds}^+(w)_u \models \neg(\sigma_i)_u$ *for some i* $(1 \leq i \leq k)$*.*

Example. In the blocks-world formalization introduced in section 1.2, the hypothesis $\Gamma =$ `(ON ?X ?*Y)` obviously satisfies verification condition (2) of Definition 3 (no simultaneous multiple effects), because the two (secondary) *when*-clauses of `Put` have only one `ON`-effect with the same pair of parameters. Moreover, condition (3.a.i) holds for both such effects, because w_1 (the primary *when*-clause of `Put`) has preconditions `(ON ?x ?z)` and `(NEQ ?y ?z)`, and `(NOT (ON ?x ?z))` is an effect of both the secondary *when*-clauses of `Put`. Thus, if Γ also satisfies condition (1) of Definition 3, Γ is a valid state constraint.

As in the case of implicative constraints, these verification conditions are not used directly, but rather, we start with a set of candidate supplementary conditions and then use violations of criteria (2) and (3a(i)) to identify possible excuses for those violations in accord with clauses (2a,b), (3a(ii)) and (3b).[9]

[8] We include the inequality for clarity here, but it is redundant as long as operators are assumed not to produce contradictory effects.

[9] See [16] for discussion of the subroutines on which testing of sv-constraints depends. For criterion (1) the relevant routine is `initially-refute-sv-constraint`. For (2) it is `test-sv-effect` and for (3) it is `test-concomitant-change`.

As usual the proofs of the following correctness and complexity theorems are given in Appendix A.

Theorem 4 *If the verification conditions of Definition 3 for sv-constraints hold for an sv-constraint* $(\phi^*\ \sigma_1...\sigma_k)$*, then the constraint holds in all reachable states for all values of its variables. (Hence sv-constraints found by* DISCOPLAN *'s "dedicated" method for such constraints are correct.)*

Theorem 5 *The "dedicated" computation of sv-constraints in accordance with the hypothesize-and-test scheme is completed in polynomial time for any bound* max *on the allowable number of supplementary conditions, specifically in*

$$O\left(\sum_{o\in O} l_o^4 \left[\sum_{o'\in O} l_{o'}^3 \ +\ \frac{max}{l_o}\cdot\sum_{o'\in O} l_{o'}^{max+2} \ +\ n_{init}^2\right]\right)$$

steps, where O is the set of planning operators, l_o *is the size of operator o, and* n_{init} *is the number of initial conditions.*

4.3 Implicative Constraints + Single-Valuedness Constraints

We now successively relax two of the syntactic requirements defining a simple implicative constraints ($(\phi \Rightarrow \psi)\sigma_1...\sigma_n$): first, we drop the requirement that if ϕ contains variables not occurring in ψ, then ψ must be upward monotonic; then we drop the requirement that the variables of ϕ must include all those occurring in ψ. As explained earlier, these more general constraints require simultaneous confirmation of certain sv-constraints.

4.3.1 The Case of Subsumed Variables

In the introduction, we mentioned the blocks-world constraint

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

as an example of a combined implicative and sv-constraint obtainable by 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, whenever the supplementary conditions hold.

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)`, consequent `(Q ?X)`, and supplementary condition `(S ?X)`, the verification conditions are the following. These are readily generalized to allow for multiple shared and starred variables and for constants; `(S ?X)` can then be reinterpreted as a conjunction of multiple (positive or negative) supplementary conditions on multiple unstarred variables occurring in the implication.[10] Also the implicative

[10] Allowing for starred variables in supplementary conditions would require a special postprocessing step in constraint verification; however, though we allow for supplementary conditions involving starred variables

consequent may be negated. The *when*-clause variable w is understood to range over all *when*-clauses of all operators.

Definition 4 (Verification conditions for an implicative + sv-constraint) *Let* $\Gamma =$ `((IMPLIES (P ?X ?*Y) (Q ?X)) (S ?X))`. DISCOPLAN*'s verification conditions for* Γ *are:*

(1) The implicative constraint is true initially, for all substitutions of constants in the planning domain for variables of the constraint.

(2) (No simultaneous multiple effects) There must not be multiple effects matching `(P ?X ?*Y)` *that could directly violate the sv-constraint. The details are as in condition (2) for "dedicated" sv-testing.*

(3) If w *contains an effect* `(P` x y`)` *for some parameters or constants* x *and* y*, then either*

(a) (i) (Concomitant effect or persistent precondition) there is an effect or persistent precondition `(Q` x'`)` *where* $x' =_w x$*; and (ii) (No cumulative multiple effects) there is either a precondition* `(NOT (Q` x'`))` *where* $x' =_w x$*, or a change that "compensates" for the effect* `(P` x y`)`*, i.e., a precondition* `(P` x' z`)` *and an effect* `(NOT (P` x'' z'`))`*, with* $x'' =_{w,w_2} x' =_w x$ *and* $z' =_{w,w_2} z$*, and if* $w_2 \neq w, w_1$ *then* `(S` x`)` $\models_w$ preconds(w_2)*;*[11] *or else*

(b) preconds$^+(w) \models$ `(NOT (S` x`))`*.*

(4) If w *contains an effect* `(NOT (Q` x`))` *for some* x*, then either*

(a) (Concomitant change) there is a precondition `(P` x' y`)` *in* preconds$^+(w)$*, and an effect* `(NOT (P` x'' y'`))` *in some* when*-clause* w_2*, where* $x'' =_{w,w_2} x'$ *and* $y =_{w,w_2} y'$*, and if* $w_2 \neq w, w_1$ *then* `(S` x'`)` $\models_w$ preconds(w_2)*; or else*

(b) preconds$^+(w) \models$ `(NOT (S` x`))` y*.*

We take the last condition to mean that `(S` x`)` (or, in the generalized case, one of the several conditions this may stand for) is contradicted (via `EQ/NEQ`-preconditions of w and w_1 and possibly type constraints) by some element of *preconds*$^+(w)$.

Example. In the blocks-world formalization introduced in section 1.2, the hypothesis $\Gamma =$ `((IMPLIES (ON ?*X ?Y) (NOT (CLEAR ?Y))) (NEQ ?Y TABLE))` satisfies verification condition (2) of Definition 4 (no simultaneous multiple effects), because the second arguments of effect `(ON ?x ?y)` in the first (secondary) *when*-clause w of `Put` and effect `(ON ?x ?y)` in the other secondary *when*-clause w_2 are constrained to be different by the `EQ/NEQ`-preconditions of w and w_2 (see condition 2 of Definition 3, for which we disprove that the unstarred variable `(ON ?*X ?Y)` can be bound to the same object when unified with the two `ON`-effects of w and w_2).

Moreover, Γ satisfies condition (3) of Definition 4 because, for the effect `(ON ?x ?y)` of w condition (3.b) holds (i.e., preconds$^+(w) \models$ (`EQ ?y TABLE`)), while for the same effect of w_2 condition (3.a) holds (because `(NOT (CLEAR ?y))` is an effect of w_2). Finally, Γ satisfies condition (4.a) of Definition 4 because, for the effect `(CLEAR ?z)` appearing in both w and w_2, both preconds$^+(w)$ and preconds$^+(w_2)$ contain `(ON ?x ?z)`, and `(NOT (ON ?x ?z))` is an

in forming hypotheses, we have not come across significant constraints with such supplementary conditions in our experimentation in a variety of planning domains.

[11] Part (ii) involves use of `test-compensating-change`, detailed in [16].

effect of both w and w_2. Thus, if condition (1) of Definition 4 holds as well, then Γ is a valid state constraint.

The motivation for conditions (1) and (2) should be fairly self-evident. In condition 3(a)(ii), however, the reason for checking whether `(NOT (Q` x'`))` (with $x' = x$) holds in the preconditions may not be entirely obvious. The reasoning is that such a condition implies universal falsity of `(P` x z`)` for all values of z in the prior state, assuming that the implicative constraint holds in that state; this in turn ensures that the assumed effect `(P` x y`)` does not subvert single-valuedness of `P` in its second argument. The second alternative in 3(a)(ii) ensures that if universal falsity of `(P` x z`)` in the prior state cannot be established, then there is a change from a true instance to a false instance of `P` that compensates for the sv-threatening effect `(P` x y`)`. (3b) is an "escape clause", allowing violation of the implicative and sv-parts of the constraint for values of the variables that do not satisfy the supplementary conditions. In (4), the reason for requiring an effect `(NOT (P` x'' y'`))` (under the equality $x'' = x$) is to maintain the truth of the implication. Its truth needs to be maintained for all values of y', and this is assured if `(P` x' y`)` (with $x' = x''$, $y = y'$) holds in the prior state, assuming single-valuedness of `P` in its second argument in the prior state.

The following simple example serves both to illustrate the role of verification conditions 3(a) and 4(a), and the way in which a hypothesis being tested may enter inductively into the verification process. The available operators describe two ways of getting from one place to another – walking and taking a cab:

```
(action walk                                (action take-cab
 :parameters (?x ?y)                         :parameters (?x ?y)
 :precondition  (and (at ?x) (neq ?x ?y))    :precondition (and (at-cab ?x) (neq ?x ?y))
 :effect (and (at ?y) (not (at ?x)))         :effect (and (at-cab ?y) (not (at-cab ?x)))

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

This is an implicative + sv-constraint, with no new variables (in fact, no variables at all) in the consequent. Consider first the `Get-out` operator, whose effect `(AT ?x)` brings into play verification condition 3(a). Part (i) is satisfied because of the effect `(NOT (IN-CAB))`, maintaining the truth of the implication. Part (ii) is satisfied because of the precondition `(IN-CAB)`, which ensures falsity of the `AT`-predicate for all arguments, in the prior state. Next, consider the `Get-in` operator, whose effect `IN-CAB` brings into play verification condition 4(a) (since `IN-CAB` negates the consequent of the implication). The condition is evidently met because of the concomitant change from `(AT ?x)` to `(NOT (AT ?x))`. Finally, the `Walk` operator has an effect `(AT ?y)`, again bringing into play condition 3(a). Part (ii) is met because of the compensating change from `(AT ?x)` to `(NOT (AT ?x))`, maintaining single-valuedness of `AT`. But for part (i), we appear to lack an effect or persistent precondition `(NOT (IN-CAB))`, needed to maintain the truth of the implication. Thus the verification conditions, applied directly to the operators as we have defined them, do not suffice to confirm the invariant.

One rather unsatisfactory remedy would be to add `(NOT (IN-CAB))` manually as a precondition of `Walk`. This is justifiable in principle since `(NOT (IN-CAB))` is a consequence of precondition `(AT ?x)` *and the invariant under consideration.* But we constructed this example specifically to show that automatically augmenting preconditions with consequences of the hypothesis being tested can be useful in certain cases. Our full set of methods, which include the bootstrapping methods mentioned earlier, actually discover the invariant, albeit relatively slowly.

Theorem 6 *An implicative state constraint* `((IMPLIES (P ?X ?*Y) (Q ?X)) (S ?X))` *(or generalizations allowing for more variables, a negated consequent, and multiple supplementary conditions) produced by* DISCOPLAN *is correct.*

The proof, and indications of how we generalize to allow for multiple starred and unstarred variables, are given in Appendix A.

4.3.2 The Case of Non-Subsumed Variables: Exclusive State Constraints

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. We will consider the special case of constraints of form

```
((IMPLIES (P ?X ?*Y) (NOT (Q ?X ?*Z))) (S ?X)).
```

Again, this is readily generalized to cases where `?X` and `?*Y` stand for multiple unstarred and starred variables, constants as well as variables may occur in `(P ...)` and `(Q ...)`, and `(S ?X)` may consist of multiple static constraints on unstarred variables. 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). The discovery of such constraints proceeds much as in the case of implicative constraints with subsumed variables (previous subsection), and we will only point out the main differences. 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 that restrictions relevant to exclusive state constraints are placed on signs and variables (i.e., the antecedent is positive, the consequent is negative, and each contains variables not occurring in the other). There are six verification conditions, which run as follows.

Definition 5 (Verification conditions for an exclusive state constraint) *Let* $\Gamma =$ `((IMPLIES (P ?X ?*Y) (NOT (Q ?X ?*Z))) (S ?X))`. DISCOPLAN*'s verification conditions of* Γ *are:*

(1) The exclusive state constraint (including the sv-constraints) is true initially, for all substitutions of constants in the planning domain for variables of the constraint.

(2) (No simultaneous multiple `P`*-effects) There must not be multiple effects matching* `(P ?X ?*Y)` *that could directly violate the sv-constraint, or else some supplementary*

condition (one of those symbolized by `(S ?X)`*) must be demonstrably false. The details are as in condition (2) for "dedicated" sv-testing.*

(3) (No simultaneous multiple `Q`*-effects) Similarly, there must not be multiple effects matching* `(Q ?X ?*Z)` *that could directly violate the sv-constraint, or else some supplementary condition must be demonstrably false.*

(4) If an operator o has a when*-clause w with an effect* `(P x y)` *for some* x, y*, then either*

(a) (Concomitant change) there is a precondition `(Q x' z)` *in* $\text{preconds}^+(w)$*, and an effect* `(NOT (Q x'' z'))` *in some* when*-clause* w_2 *of o where* $x'' =_{w,w_2} x' =_w x$ *and* $z =_{w,w_2} z'$*, and if* $w_2 \neq w, w_1$*, then* `(S x)` $\models_w \text{preconds}(w_2)$*; or*

(b) (False supplementary conditions) $\text{preconds}^+(w) \models$ `(NOT (S x))`*.*

(5) This is the analogue of the previous condition, with `P` *and* `Q` *and y and z interchanged.*

(6) If an operator o has a when*-clause w with an effect* `(P x y)` *for some* x, y*, and some (not necessarily different)* when*-clause* w_2 *of o has an effect* `(Q x' z)` *where the* `EQ/NEQ`*-preconditions of w,* w_1 *and* w_2 *do not entail* $x \neq x'$*, then either*

(a) $\text{preconds}^+(w) \models$ `(NOT (S x))`*; or (b)* $\text{preconds}^+(w_2) \models$ `(NOT (S x'))`*.*

The motivation for conditions (1-3) is again self-evident. Condition (4a) calls for a change from (roughly) `(Q x z)` to `(NOT (Q x z))`, concomitant with the given effect `(P x y)`. This condition corresponds to condition 3(a) in the verification conditions for the subsumed-variable case in the previous subsection. The reader may wonder, however, why the two subconditions previously labeled (i) and (ii) collapse to a single "concomitant change" condition. The reason is as follows (simplifying a little by using identical names for terms that might require `EQ`-preconditions to be unified). Subcondition (i), in the present case, would be that there is an effect or persistent precondition `(NOT (Q y z))`, preserving the truth of the implication. However, this effect or persistent precondition must be assured for *all* substitutions of constants for z, since the variables of the implicative hypothesis are universally quantified. Now, the only way a persistent precondition `(NOT (Q y z))` could provide this guarantee is if z were universally quantified in the preconditions; but we are not allowing for universally quantified preconditions here, so this is not a viable alternative. The remaining alternative is that there is an *effect* `(NOT (Q y z))`. But in this case as well, we need to be sure that in the resultant state the formula holds for *all* substitutions for z. This can be assured if `(Q y z)` held in the preconditions, for then, by the induction assumption, `(Q y z)` held *only* for z (some particular constant, for any given instance of the operator), and so when this becomes false because of effect `(NOT (Q y z))`, it becomes false for all substitutions for z. But then we are positing a concomitant change from `(Q y z)` to `(NOT (Q y z))` – just as the present condition 4(a) does. Furthermore, part (ii) then becomes redundant, since the truth of `(Q y z)` in the preconditions ensures by the induction assumption that `(P x' y)` is false in the prior state for all substitutions for x', so that the effect `(P x y)` (at which condition 4(a) is aimed) will not invalidate the single-valuedness of `P` in its first argument (as long as there are no *simultaneous* multiple effects, as per condition 2).

Condition (6) 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 $x = x'$ cannot be

ruled out by EQ/NEQ-preconditions. One way to understand this condition is to think of an exclusive state constraint as a disjunction of two negative literals, and to recognize 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. Put a little differently, the effect (P x y) needs to be offset by (NOT (Q x v) becoming true for all v (as condition (4a) tries to ensure), but this would be subverted by any effect of form (Q x z).

Theorem 7 *Any exclusive state constraint* ((IMPLIES (P ?X ?*Y) (NOT (Q ?X ?*Z))) (S ?X)) *produced by* DISCOPLAN *is correct.*

Again, the proof and indications of how we generalize to allow for multiple starred and unstarred variables, constant arguments, and multiple supplementary conditions are given in Appendix A, where we also prove the following complexity result:

Theorem 8 *The computation of exclusive state constraints in accordance with the hypothesize-and-test scheme is completed in polynomial time for any bound* max *on the allowable number of supplementary conditions.*

5 Conclusions

We have provided for the first time full details of the essential concepts and algorithms underlying the well-known DISCOPLAN system for constraint (domain invariant) discovery, including theorems (with proofs) concerning the soundness of the algorithms and their complexity analysis. In doing so we have focused on the basic types of state constraints supported by DISCOPLAN, likely the most useful for planning (both historically and in future), namely type constraints, implicative constraints, single-valued constraints and exclusiveness constraints. In the fast-paced world of AI, detailed algorithms and their certification are often omitted. But we believe that it was essential to provide these in the case of DISCOPLAN, because of its exceptionally broad coverage of constraint types and unusual ability to deal directly with conditional effects, and because the requisite variants of the inductive hypothesize-and-test paradigm may appear so complex as to be untrustworthy, or of prohibitive computational complexity.

An experimental analysis presented in [16] and additional tests with recent domains (Termes and Nurikabe) show that known benchmark domains have from a few to several dozens of irredundant invariants of the basic types considered in this paper, that DISCOPLAN correctly infers (e.g., 5 in blocks-world with conditional effects, 7 in Hanoi, 8 in Termes, 13 in Nurikabe, 19 in Satellite, 24 in Logistics, 85 in Rovers, 107 in Airport).

The utility of invariant discovery in planning domains has never been in question – we previously noted their early use in deductive planning [15] – and has been verified many times; and a hypothesize-and-test paradigm, with "testing" consisting of proofs by mathematical induction, is certainly a natural one. In the context of "strong AI", our particular methods – which were seen to be quite dependent on the structure and semantics of operators and on the forms of invariants sought – pose an interesting meta-challenge for future work: How can reasoning about dynamic worlds, ones that are changed by deliberate goal-directed actions be (efficiently) automated in a uniform logical way? Humans not only are quick to detect regularities in puzzles, games, and real-world domains, but also see their

truth as "obvious". For example, someone presented with the Towers of Hanoi puzzle easily recognizes that, given the stipulated constraints on actions, disks on any peg will always be ordered by size. A rare voice pointing out the remarkable facility of human "cognitive judgements" that appear to require mathematical induction was that of David McAllester [31].

As noted at the outset, the approach and techniques described in this paper are the essential basis for further techniques implemented in DISCOPLAN, allowing discovery of additional invariants, including several types that no other existing system infers. (Some of the further techniques are outlined in [21].) Details of these techniques will be the subject of a separate paper, currently available as a technical report [16].

Acknowledgments

This research was supported in part by NATO Collaborative Research Grant CRG951285, and by ARPA/SSTO grant F30602-95-1-0025 and DARPA grants F30602-98-2-0133 and W911NF-15-1-0542 (second author). We would like to thank Fabrizio Morbini for his valuable help in the construction of the on-line version of DISCOPLAN, as well as in the implementation of a few parts of the system.

This work originated in the context of a collaboration between the Dept. of Computer Science of the University of Rochester and IRST in Trento, where the first author worked in a research group lead by Oliviero Stock who guided the start of his career in AI. We deeply thank Oliviero who initiated and supported this collaboration, initially focused on temporal reasoning and then extended to AI planning. This led to a very fruitful joint work between the authors of this paper, that has continued for several years.

6 Appendix A: Proofs of Theorems

As a preliminary to the proofs of the theorems, we provide a formalization of the notion of *reachable* states, and of the notions on which this depends. We will use the term *ground atom* in the sense of a predication whose arguments are constants (i.e., we do not consider functional terms), and correspondingly a *ground literal* is an unnegated or negated ground atom.

Definition 6 (Planning domain) *A* planning domain *consists of a finite set of PDDL operators each of which represents a STRIPS operator possibly augmented with negated preconditions,* `EQ/NEQ`*-constraints, typed parameters, and conditional effects (as described and illustrated in section 1.2).*

Recall that operators have a *primary when*-clause and possibly one or more *secondary when*-clauses (representing conditional effects), each with its own preconditions and effects, and these preconditions may include `EQ/NEQ`-constraints. All preconditions and effects are function-free literals over constants and parameters of the operator, and `EQ/NEQ`-preconditions use only the positive forms of `EQ` and `NEQ`.

Definition 7 (Planning domain states) *A* state *of a planning domain is any finite set of ground atoms. The reserved predicates* `EQ`/`NEQ` *do not appear explicitly in states.*

Note that EQ(c, c) (i.e., $c = c$) and NEQ(c, c') (i.e., $c \neq c'$) are regarded as tacitly present in every state for every constant c and every constant c' distinct from c. Also note that in principle we allow states to contain predicates that do not appear in any operator – these will simply remain invariant under any sequence of operator applications.

Definition 8 (Operator instance) *Let o be a planning domain operator with m parameters $\underline{r} = r_1, ..., r_m$ and let $\underline{d} = d_1, ..., d_m$ be an m-tuple of constants such that the substitution $\underline{d}/\underline{r}$ for the parameters of o yields no primary preconditions of form* NEQ(c, c) *or* EQ(c, c') *for distinct constants c, c'. Then the result of that substitution, written $o(\underline{d})$, is an* instance *of operator o. Note that all preconditions and effects of $o(\underline{d})$ are ground literals.*

The non-violation of EQ/NEQ-preconditions required by definition 8 removes from consideration operator "instances" that are not applicable in any state,

Definition 9 (Satisfaction of preconditions) *Let preconds$(w)_u$ be a set of ground literals obtained by applying ground substitution $u = \underline{d}/\underline{r}$ to the preconditions of when-clause w of an operator with parameters $\underline{r}$. Then preconds$(w)_u$ are* satisfied *by a state s iff every unnegated, non-*EQ/NEQ *literal of preconds$(w)_u$ is in s, no complement (unnegated version) of a negated literal of preconds$(w)_u$ is in s, and preconds$(w)_u$ contains no literals* NEQ(c, c) *or* EQ(c, c') *for any distinct constants c, c'.*

Definition 10 (Operator application and resulting state) *Let s be a state and let $o(\underline{d})$ be an operator instance obtained from o via substitution $u = \underline{d}/\underline{r}$, where the primary preconditions of $o(\underline{d})$ are satisfied by s. Then o (and $o(\underline{d})$) is* applicable *in state s and the* result *of its application, written $o(\underline{d})(s)$, is given by the following additions to, and deletions from, s:*

1. *For the primary when-clause w_1 of $o(\underline{d})$,*
   (a) *add the positive effects of w_1 to s, and*
   (b) *delete the predications from s whose negations appear as effects of w_1;*
2. *For each secondary when-clause w of $o(\underline{d})$ such that preconds$(w)_u$ are satisfied by (the original) s,*
   (a) *add the positive effects of w to s, and*
   (b) *delete from s the predications whose negations appear as effects of w.*

There is the usual problem here that some "additions" and "deletions" might be in conflict, adding and taking away the same atom. We could assume that deletions are performed before additions, i.e., additions win over deletions. But we generally make the stronger assumption that operators are designed to entirely avoid such inconsistent effects. Obviously, our definition implements the familiar *STRIPS assumption*, that the only effects of an operator are those it explicitly asserts. We are now able to define *reachable* states:

Definition 11 (Reachable states) *A state s is* reachable *from a given initial state s_0 in a given planning domain O iff there is a sequence of operator instances $o_1(\underline{d}^{(1)}), ..., o_k(\underline{d}^{(k)})$, where $\{o_1, ..., o_k\} \subseteq O$, and a corresponding sequence of states $s_0, ..., s_k$, such that $s = s_k$, and for $i = 1, ..., k$, o_i is applicable in s_{i-1} and $o_i(\underline{d}^{(i)})(s_{i-1}) = s_i$.*

We now turn to the proofs of the theorems. (Theorem 1 concerning Find-type-constraints was established in the main text.)

The following lemma establishes a relationship between unifiers obtained by matching hypothesis literals to operator effects or preconditions and the corresponding unifiers obtained by matching those literals to ground instances of the effects or preconditions. The lemma will facilitate the proof of correctness of the verification conditions for several types of constraints found by the hypothesize-and-test method. Where we speak of *constrained unifiers*, we refer tacitly to some given set of `EQ/NEQ`-constraints, relating the parameters occurring in an effect to each other or to constants. A constrained unifier must not equate two terms (directly, or through a chain of unifications) if either (i) these terms belong to two equality sets $\mathcal{E}_1$, $\mathcal{E}_2$ (as determined by the `EQ`-constraints) where for some $\tau_1 \in \mathcal{E}_1$ and $\tau_2 \in \mathcal{E}_2$, there is a `NEQ`-constraint between τ_1 and τ_2; or (ii) one of these terms is a constant, and the other equals another constant *via* `EQ`-constraints. (Of course, if the terms are distinct constants, then even the unconstrained unification fails.)

Unification Lemma. Let $\underline{X}$ be an m-tuple of variables and constants, $\underline{x}$ be an m-tuple of parameters and constants, and $\underline{b}$ be an m-tuple of constants.

Suppose $\underline{x}$ and $\underline{b}$ have constrained unifier v, and $\underline{X}$ and $\underline{b}$ have unifier v'. Then $\underline{X}$ and $\underline{x}$ have constrained unifier u, the composition uv is also a constrained unifier, $\underline{X}_{uv} = \underline{X}_{v'} = \underline{b}$, and $\underline{x}_{uv} = \underline{x}_v = \underline{b}$.

Remarks. We can interpret a unifier u as a set $|u|$ of equivalence classes of size > 1, where each u-equivalence class contains a set of directly or indirectly equated (unified) terms, distinct from the terms in other equivalence classes. Note that such an equivalence class may contain at most one constant, and any number of parameters and variables.[12] The composition uv of two unifiers can be thought of as the union of equations comprising u and v. Correspondingly, $|uv|$ is obtained by repeatedly merging non-disjoint equivalence classes from $|u|$ and $|v|$ until we again have a set of (one or more) disjoint classes. However, if any of the resulting equivalence classes contain multiple constants, we say that uv does not exist (more accurately, is inconsistent). Note that a set of `EQ`-constraints also induces a set of equivalence classes on the equated terms, and so we can regard a unifier u constrained by a set $\mathcal{C}$ of `EQ/NEQ`-constraints as tacitly composed with the given `EQ`-constraints. But in this case consistency (existence) requires not only that each equivalence class of $|u|$ contain at most one constant, but also that no two terms of an equivalence class be related by a `NEQ`-constraint. We will refer to such unifiers as $\mathcal{C}$-*consistent*.

In these terms, the lemma states that whenever v and v' are $\mathcal{C}$-consistent, so are u and uv; furthermore any variable X occurring in $\underline{X}$ will be in the uv-equivalence class of a constant b occurring in $\underline{b}$ just in case it is in the v'-equivalence class of b; and any parameter x occurring in $\underline{x}$ will be in the uv-equivalence class of a constant b occurring in $\underline{b}$ just in case it is in the v-equivalence class of b.

[12] However, for the unifiers u, v, v' of the lemma an equivalence class cannot consist entirely of variables or entirely of parameters. The former is the case because in our asymmetric form of unification, where a parameter can be substituted for a variable but not *vice versa*, and where variables can occur only in the hypothesis literal, variables can become equated only by having the same constants or parameters substituted for them. The latter is the case because parameters occur only in $\underline{x}$, so that two parameters can become equal only *via* equality to a constant or variable occurring in $\underline{X}$ or $\underline{b}$.

Proof. Suppose that v and v' exist (are $\mathcal{C}$-consistent). Consider vv'. We claim this is $\mathcal{C}$-consistent, since it merely adds certain variables of $\underline{X}$ to the v-equivalence classes of certain constants; specifically, if X_i is a variable, then it is added to the v-equivalence class of b_i (since X_i and b_i are in the same v'-equivalence class). It is not possible that the same variable is added to the equivalence classes of two different constants (thus forcing a merger of two v-equivalence classes) because this could only happen if the v'-equivalence class of that variable contained two constants, which is impossible by the consistency of v'. So, since furthermore `EQ/NEQ`-constraints don't involve any variables, vv' is $\mathcal{C}$-consistent.

We now claim further that $|uv| = |vv'|$. This is because for each i ($1 \leq i \leq m$), uv contains just the equations $X_i = x_i$ and $x_i = b_i$ while vv' contains just the equations $x_i = b_i$ and $X_i = b_i$, and this obviously leads to the same equivalence classes for uv and vv'. From this the $\mathcal{C}$-consistency of uv and the consistency of u follow (since $|u|$ is a refinement, in a set-partitioning sense, of $|uv|$); it also follows that uv equates variables of $\underline{X}$ to the same constants as v'; and finally it follows that uv equates parameters of $\underline{x}$ to the same constants as v (for the latter the $\mathcal{C}$-consistency of uv suffices). □

Corollary 1. Let the premises be as in the Unification Lemma. Then for any sequence $\underline{X}'$ whose components are variables of $\underline{X}$ and constants,

$\underline{X}'_{uv} = \underline{X}'_{v'}$,

and for any $\underline{x}'$ whose components are parameters of $\underline{x}$ and constants,

$\underline{x}'_{uv} = \underline{x}'_{v}$.

Proof. Immediate from the Unification Lemma. □

The following corollary slightly strengthens the Unification Lemma.

Corollary 2. Let $\underline{X}$, $\underline{x}$ and $\underline{b}$ be as in the Unification Lemma. Suppose that v is a constrained unifier such that $\underline{x}_v = \underline{b}$, where v may involve parameters other than those occurring in $\underline{x}$ (but no variables), and v' is a unifier such that $\underline{X}_{v'} = \underline{b}$, where v' may involve variables other than those occurring in $\underline{X}$ (but no parameters). Then $\underline{X}$ and $\underline{x}$ have a constrained unifier u, the composition uv is also a constrained unifier, and $\underline{X}_{uv} = \underline{X}_{v'} = \underline{x}_{uv} = \underline{x}_v = \underline{b}$.

Proof. Let v_1 be the result of reducing v to a substitution of constants for the parameters of $\underline{x}$. (In terms of equivalence classes, we delete parameters occurring in v but not in $\underline{x}$ from their equivalence classes in $|v|$, and then extract the substitutions of constants for the remaining parameters.) Similarly let v'_1 be the result of reducing v' to a substitution of constants for the variables of $\underline{X}$. Clearly v_1 is still a constrained unifier and $\underline{x}_v = \underline{b}$, and v'_1 is still a unifier and $\underline{X}_{v'_1} = \underline{b}$.

But then $\underline{X}$, $\underline{x}$, $\underline{b}$, v_1, and v'_1 satisfy the conditions of the Lemma and so $\underline{X}_{uv_1} = \underline{X}_{v'_1} = \underline{x}_{uv_1} = \underline{x}_{v_1} = \underline{b}$. Since v differs from v_1 only in (potentially) involving parameters that do not occur in $\underline{x}$, and since $\underline{X}_u$ cannot involve parameters other than those occurring in $\underline{x}$, $\underline{X}_{uv_1} = \underline{X}_{uv}$. Also, since $\underline{x}_u$ cannot involve parameters that do not occur in $\underline{x}$, $\underline{x}_{uv_1} = \underline{x}_{uv}$; and for similar reasons it is easy to see that $\underline{X}_{v'_1} = \underline{X}_{v'}$ and $\underline{x}_{v_1} = \underline{x}_v$. The conclusion follows. □

Preliminary remarks about correctness proofs. In the following correctness proof for the discovery of simple implicative constraints by DISCOPLAN, we show in detail the role of the Unification Lemma and its corollaries. We will not normally go into this much detail, as there is a trade-off between intuitive transparency of the reasoning and the level

of detail. Typically, our correctness arguments are structured in the following way. We consider an instance of an operator o applied in a particular state, where in that state the constraint Γ under consideration holds. We then consider the various specific ways in which the constraint might be violated in the result state. In each case, we argue that the violation would have to be produced by an effect or multiple effects of the operator instance, and that the verification conditions for Γ rule out such a violation.

These case-by-case arguments are intuitively quite clear, if one thinks about the instance of o under consideration *as if* the verification conditions applied directly to operator instances. But in fact, verification conditions apply to parametrized operators, and this leads to some hidden subtleties in this mode of argumentation. Strictly, we have to argue that the occurrence of certain ground literals as effects of an operator instance, applied in a state satisfying certain conditions, entails the existence of certain parametric precondition and effect literals in the operator; further, these precondition and effect literals will unify with certain literals of Γ, and hence specific verification conditions will come into play, and impose requirements on various unification instances of the preconditions and effects of o; and these unification instances in turn must be shown to reduce to ground literals relevant to the operator instance, allowing completion of the *reductio* argument. It is the "ascent" from specific ground literals to parametric operator literals, and the "descent" from unification instances of the latter back down to the ground level, that the Unification Lemma is designed to facilitate. These moves are nontrivial because of the subtleties of unification involving both Γ-variables and operator parameters, and `EQ/NEQ`-preconditions constraining the latter. Still, as mentioned, such details can be distracting and we will generally suppress them, except in the case of simple implicative constraints, sv-constraints discovered by the "dedicated" method, and n-ary disjunctive + sv-constraints. In the last of these constraint types, the complexity of the details is such as to motivate a Verification Lemma (for the particular verification conditions concerned) showing that we can in fact argue as if the verification conditions applied directly to operator *instances*.

Theorem 2 *If a simple implicative constraint* Γ *holds initially, and the verification conditions of Definition 2 for such constraints are met, then* Γ *holds in all reachable states. (Hence simple implicative constraints found by* DISCOPLAN *are correct.)*

Proof. Assume for induction that Γ holds prior to the application of an instance $o(\underline{d})$ of an operator $o(\underline{r})$, where $\underline{r}$ is the list of parameters of o. We will show that Γ still holds after application of $o(\underline{d})$. For clarity in the use of the Unification Lemma, we abbreviate the substitution $\underline{d}/\underline{r}$ as v. Note that v necessarily satisfies the `EQ/NEQ`-constraints of the primary *when*-clause w_1 of o, and of every other *when*-clause w whose preconditions are met when $o(\underline{d})$ is applied. In other words, v is a constrained unifier relative to those `EQ/NEQ`-constraints.

Since Γ holds prior to application of o, therefore for every substitution v' of constants (in the planning domain) for variables of Γ, at least one of $\overline{\phi}_{v'}$, $\psi_{v'}$, $(\overline{\sigma}_1)_{v'}$, ..., $(\overline{\sigma}_k)_{v'}$ holds in the prior state. We show for each case in turn that $\Gamma_{v'}$ holds in the resultant state.

Case 1: $\overline{\phi}_{v'}$ holds in the prior state. If $\overline{\phi}_{v'}$ still holds in the resultant state, then $\Gamma_{v'}$ still holds as well. So assume $\overline{\phi}_{v'}$ is false in the resultant state, i.e., $\phi_{v'}$ is true in it. Then by the semantics of operators, there must be some *when*-clause w of o such that $preconds^+(w)_v$ hold in the prior state and which has an effect ϕ' such that $\phi'_v = \phi_{v'}$. Then with $\underline{X}$ = argument list of ϕ and $\underline{x}$ = argument list of ϕ', Corollary 2 of the Unification Lemma

applies, so that ϕ' must be unifiable with ϕ with some unifier u (constrained by the EQ/NEQ-conditions of w and w_1) such that $\phi'_{uv} = \phi'_v = \phi_{v'}$, where uv is also a constrained unifier. Hence by verification condition (2) one of the following holds: w or w_1 has (a) an effect or (b) a w-persistent precondition ψ' such that $\psi'_u \equiv_{w_1,w} \psi_u$; or (c) o has a *when*-clause w_2 ($\neq w, w_1$) such that $(\sigma_1)_u, ..., (\sigma_k)_u \models \mathit{preconds}(w_2)_u$ and w_2 has an effect ψ' such that $\psi'_u \equiv_{w_1,w_2} \psi_u$; or (d) $\mathit{preconds}^+(w)_u \models \neg(\sigma_i)_u$, for some i ($1 \leq i \leq k$). But then each of (a-d) maintains the truth of $\Gamma_{v'}$. In particular the effect ψ' posited by (a) satisfies $\psi'_{uv} = \psi_{uv}$, since all (ground) instances of two constrained-equivalent formulas are identical. But by Corollary 1 of the Unification Lemma $\psi'_{uv} = \psi'_v$ and $\psi_{uv} = \psi_{v'}$, so that for the instance $o(\underline{d})$ of o, the effect ψ'_v is identical with $\psi_{v'}$. This assures the truth of $\Gamma_{v'}$ in the resultant state. For (b) the argument is similar, except that we also need to address the case where the persistence of ψ'_v is threatened by an effect $\overline{\psi}''$ of some other secondary *when*-clause w_2, whose preconditions also happen to hold, and where $\psi''_v = \psi_{v'}$. As already noted, $\psi_{v'} = \psi_{uv}$, so $\psi''_v = \psi_{uv}$; hence applying Corollary 2 of the Unification Lemma to the argument lists of ψ'' and ψ_u (with v viewed as the unifier of both of those lists with the resulting identical lists of constants), we know that ψ'' and ψ_u are unifiable, with some unifier u' (which also entails that uu' is a unifier constrained by the EQ/NEQ conditions of w and w_2); furthermore, $u'v$ is a unifier constrained by the EQ/NEQ conditions of w and w_2 (keeping in mind that v is a unifier constrained by the EQ/NEQ conditions of w), $\psi''_{u'v} = \psi''_v$, and $\psi_{uu'v} = \psi_{uv} = \psi_{v'}$. So by the details of verification condition 2(b), $\mathit{preconds}^+(w_2)_{uu'} \models \neg(\sigma_i)_{uu'}$, for some i ($1 \leq i \leq k$). Hence $\mathit{preconds}^+(w_2)_{uu'v} \models \neg(\sigma_i)_{uu'v}$. But $\mathit{preconds}^+(w_2)_{uu'v} = \mathit{preconds}^+(w_2)_{uv} = \mathit{preconds}^+(w_2)_v$ (by two applications of Corollary 1 of the Unification Lemma, first with the arguments of $\mathit{preconds}^+(w_2)_u$ in the role of $\underline{x}'$, then with the arguments of $\mathit{preconds}^+(w_2)$ in that role) and $\neg(\sigma_i)_{uu'v} = \neg(\sigma_i)_{v'}$ (again by two applications of Corollary 1). Then $\mathit{preconds}^+(w_2)_v \models \neg(\sigma_i)_{v'}$, so that $\Gamma_{v'}$ is again true in the resultant state. If (c) holds, we can argue much as for (a), except that we also need to ascertain that $(\sigma_1)_{v'}, ..., (\sigma_k)_{v'} \models \mathit{preconds}(w_2)_v$. This is so since in this case $(\sigma_1)_u, ..., (\sigma_k)_u \models \mathit{preconds}(w_2)_u$, and $(\sigma_i)_{uv} = (\sigma_i)_{v'}$ ($1 \leq i \leq k$) (by Corollary 1, with the arguments of σ_i in the role of $\underline{X}'$) and $\mathit{preconds}(w_2)_{uv} = \mathit{preconds}(w_2)_v$ (by Corollary 1, with the arguments of $\mathit{preconds}(w_2)$ in the role of $\underline{x}'$). If (d) holds, then $\mathit{preconds}^+(w)_u \models \neg(\sigma_i)_u$ for some i; hence $\mathit{preconds}^+(w)_v \models \neg(\sigma_i)_{v'}$, using $\mathit{preconds}^+(w)_{uv} = \mathit{preconds}^+(w)_v$ and $(\sigma_i)_{uv} = (\sigma_i)_{uv'}$, by uses of Corollary 1 much as in (b) and (c). So with $\neg(\sigma_i)_{v'}$ true, $\Gamma_{v'}$ again holds in the resultant state.

Case 2: $\psi_{v'}$ holds in the prior state. The argument for the truth of $\Gamma_{v'}$ in the resultant state is completely analogous to that for case 1, using the verification conditions (3a-c) (which mirror those of (2a-c)).

Case 3: $(\overline{\sigma}_i)_{v'}$ holds in the prior state for some i ($1 \leq i \leq k$). Then since σ_i is a static condition, $(\overline{\sigma}_i)_{v'}$ still holds in the resultant state. □

Theorem 3 *The computation of simple implicative constraints in accordance with the hypothesize-and-test scheme is completed in polynomial time for any bound* max *on the allowable number of supplementary conditions, specifically in*

$$O\left(\sum_{o \in O}\left[l_o^3\left(\sum_{o' \in O} l_{o'}^2 + \frac{max}{l_o} \cdot \sum_{o' \in O} l_{o'}^{max+2} + l_o n_{init}\right)\right]\right)$$

steps, where O is the set of planning operators, l_o is the size of operator o, and n_{init} is the number of initial conditions.

Proof. The outer summation of the above complexity bound, and a part l_o^2 of the factor l_o^3, corresponds to the iteration over all effect-effect (or effect-precondition) combinations of all operators in the formation of hypotheses. The first inner summation reflects the checking of a hypothesis against all preconditions and effects (contributing $l_{o'}$) and the collection of excuses for every apparent violation. The latter phase contributes a factor $l_o l_{o'}$ per violation. This collapses (i) a factor $l_o l_{o'}$ for singleton excuses that relate an operator precondition to the negation of a candidate supplementary condition, keeping in mind that candidate supplementary conditions originated in operator o; and (ii) a similar term $(l_o l_{o'} + l_{o'})$ for collection of groups of candidate supplementary conditions that entail the preconditions of a *when*-clause and thereby ensure the generation of a "missing effect". (The $l_o l_{o'}$-part comes from the pairwise matching of preconditions to supplementary conditions, and the $l_{o'}$-part from the scan of effects for the "missing effect".) The second inner summation comes from the search for minimal sets of supplementary conditions (see footnote 6) and the third element of the sum reflects the verification of the hypothesis in the initial conditions (see the discussion of the algorithm in Figure 2, and note that l_c is bounded by l_o). □

Theorem 4 *If the verification conditions of Definition 3 for sv-constraints hold for an sv-constraint* $(\phi^*\ \sigma_1 ... \sigma_k)$*, then the constraint holds in all reachable states for all values of its variables. (Hence sv-constraints found by* DISCOPLAN *by the "dedicated" method are correct.)*

Proof. For clarity we consider the special case where $(\phi^*\ \sigma_1 ... \sigma_k)$ is

`((P ?X ?*Y) (S1 ?X ?*Y) ... (Sk ?X ?*Y))`.

The generalization to multiple "unstarred" and "starred" variables (and possibly constants) is fairly straightforward (think of `?X`, `?*Y` as sets of arguments); the remarks following the proof will provide some relevant specifics. The supplementary conditions `(Si ?X ?*Y)` of course need not depend on both `?X` and `?*Y`, but they may. In the absence of supplementary conditions, the notation `(P ?X ?*Y)` means that

$\forall x, y, z. P(x, y) \wedge P(x, z) \Rightarrow y = z$ (in all reachable states).

Let us abbreviate $S1(x, y) \wedge ... \wedge Sk(x, y)$ as $S(x, y)$. Then the sv-constraint including supplementary conditions states that

$\forall x, y, z. S(x, y) \wedge S(x, z) \wedge P(x, y) \wedge P(x, z) \Rightarrow y = z$ (in all reachable states).[13]

We can take the sv-constraint to be true in the initial state, since DISCOPLAN straightforwardly verifies this, as per verification condition (1). Given any instance of any operator o, and a prior state s_{prior} in which the preconditions of the primary *when*-clause w_1 of that operator instance are true, we assume for induction that the constraint holds in s_{prior}. We want to show that it also holds in the resultant state s_{result}. Assume otherwise, i.e., for some constants a, b, c, where $b \neq c$, we have

(C1) $S(a, b), S(a, c), P(a, b), P(a, c)$ true in s_{result}.

Since S is static, we also had

(C2) $S(a, b), S(a, c)$ true in s_{prior}.

But by the induction assumption, at least one of the literals in (C1) must have been false in s_{prior}, and so

[13] Although we will not make use of this, note that another way of writing this in our DISCOPLAN notation would be as `(P' ?X ?*Y)`, where by definition $\forall x, y. P'(x, y) \Leftrightarrow S(x, y) \wedge P(x, y)$. In other words, we are asserting the (nonstrict) sv-ness of P', which is P conjunctively augmented with the supplementary conditions.

(C3) either $P(a, b)$ or $P(a, c)$ was false in s_{prior}.

We consider the case where $P(a, b)$ was false (and $P(a, c)$ was true or false). The case where $P(a, c)$ was false is completely symmetrical and need not be separately considered. If $P(a, b)$ was false in s_{prior}, then its truth in s_{result} must be due to some effect $P(h, k)$ of some *when*-clause w of o, where h, k were respectively instantiated to a, b (if not already equal to them) and where the preconditions of w are true for the operator instance under consideration. (We have already assumed the preconditions of the primary *when*-clause w_1 to be true.) We now show three things for the operator instance under consideration: (i) there is no additional effect $P(a, c)$ of the operator instance, with $c \neq b$; (ii) w or w_1 has a precondition $P(q, r)$, which was true for $q = a$, $r = d$ for some $d \neq b$ for the operator instance, and (iii) w_1, w, or some other secondary *when*-clause w_2 generated an effect $\neg P(a, d)$, for the operator instance.

Concerning (i), if there were an effect $P(a, c)$, there would be an effect literal $P(q, r)$ of o in w or some other *when*-clause w_2 (which could be w_1 or a secondary *when*-clause) where we cannot prove that $q \neq h$, or cannot prove that $r = k$. (Keep in mind we are assuming a w-effect $P(h, k)$ instantiated as $P(a, b)$.) But then in view of the way DISCOPLAN tests `(P ?X ?*Y)` and adds supplementary conditions (i.e., by the contrapositive of verification conditions (2a,b)), there would be a supplementary condition which entails the falsity of a precondition of w, w_1, or w_2. (We will provide a more detailed version of this argument in the supplementary remarks following the proof.) But since we have assumed those preconditions to be true for the operator instance under consideration, $S(a, b)$ or $S(a, c)$ must be false in s_{prior}, contrary to (C2). So there is no effect $P(a, c)$.

Concerning (ii), suppose neither w nor w_1 has such a precondition. Then they have no precondition of form $P(q, r)$ at all, or if they do, it is not provable that $q \neq r$, or it is not provable that $r = k$. Then again from the way DISCOPLAN tests `(P ?X ?*Y)` and adds supplementary conditions (i.e., by the contrapositive of verification condition (3b)), there is a supplementary condition that entails the falsity of a precondition of w or w_1, and since these preconditions are true for the instance in question, $S(a, b)$ or $S(a, c)$ must be false in s_{prior}, contrary to (C2). So there is a precondition $P(q, r)$ as specified in (ii).

Concerning (iii), suppose there is no such effect of w or w_1. Then there is no effect of w or w_1 of form $\neg P(q', r')$ at all or for any such effect it is not provable that $q' = q$ or it is not provable that $r' = r$ (with q, r determined by (ii)). Then from verification condition (3a(ii)) (i.e., from the way DISCOPLAN tests `(P ?X ?*Y)` and adds supplementary conditions to assure a "concomitant change") and from (C2) (which precludes alternative verification condition (3b)), there is a subset of the supplementary conditions that entails the preconditions of some *when*-clause w_2 other than w or w_1, where that *when*-clause has an effect $\neg P(q', r')$, and provably $q' = q$ and $r' = r$. But then (iii) is true.

Having established the truth of (i-iii), we note that from (i) and (C1) it follows that $P(a, c)$ was true in s_{prior}. Hence by the induction assumption $P(a, y)$ was false for all $y \neq c$ in s_{prior}. Hence the constant d referred to in (ii) and (iii) is the same as c; so by (iii), $\neg P(a, c)$ holds in s_{result}, contrary to (C1). □

Supplementary remarks concerning the proof of Theorem 5:

In part (i) of the preceding proof, the appeal to verification condition (2) to infer the falsity of a precondition of w, w_1, or w_2 may seem clear enough. However, this is one of the places where some subtle details dependent on the Unification Lemma have been suppressed. It is instructive to take a closer look at this part of the argument, as an illustration of how

such suppressed details may be filled in. Strictly, the existence of effect literals $P(h, k)$ and $P(q, r)$, where we cannot prove $q \neq h$ or cannot prove $r = k$, is insufficient to support the argument based on verification condition (2). This condition also requires that each of $P(h, k)$ and $P(q, r)$ be (constrained-) unifiable with `(P ?X ?*Y)`, say with unifiers u and u' respectively, and the particularized relations $q_{u'} \neq h_u$, $r_{u'} = k_u$ (rather than $q \neq h$, $r = k$) should not both be provable. Now, the existence of unifiers u and u' is obvious for the special case of ϕ^* assumed in the proof, but note that both `EQ/NEQ`-preconditions and generalization of the proof to predicates with more than 2 arguments can complicate matters. For example, unification of `(P ?X ?X ?*Y)` with an effect `(P ?h ?k ?r)` would fail if there is a precondition `(NEQ ?h ?k)`. Of course, an instance of this effect, such as `(P a b c)`, would then not create a threat to single-valuedness of `(P ?X ?X ?*Y)` in `?*Y`. So only those instances of the effects of o that unify with ϕ^* create a threat to single-valuedness, and it suffices to ensure that the verification conditions apply to those instances. And that is precisely where the Unification Lemma helps out. Corollary 2 guarantees that if an effect literal $P(\underline{r})$ yields ground instance $P(\underline{b})$ under unifier v (where v instantiates the parameters of o), and at the same time hypothesis literal ϕ^* is unifiable with $P(\underline{b})$, with unifier v', then the unifier u of ϕ^* with $P(\underline{r})$ also exists; this guarantees that verification condition (2) applies to all relevant effect literals. Furthermore, Corollary 2 also assures us that we can compose u with v, and (returning to the special case considered in the proof) that $h_{uv} = h_v$; and similarly we have $q_{u'v} = q_v$ for the second effect literal $P(q, r)$. So, since $h_v = q_v = a$, it follows that the relation $h_u \neq q_{u'}$ cannot be provable. Similarly we can argue that the relation $k_u = r_{u'}$ cannot be provable. At this point we have established incontrovertibly that (a) or (b) in verification condition (2) must hold. But it takes one more application of Corollary 2 to establish that a relevant instance of (a) or (b) holds. In the case of (a), i.e., if the truth of $(\sigma_i)_u$ entails falsity of $preconds^+(w)_u$, then $(\sigma_i)_{v'}$ (a ground instance of σ_i) entails falsity of the preconditions $preconds^+(w)_v$ of the *instance* of o under consideration, in virtue of the equivalence of $(\sigma_i)_{uv}$ with $(\sigma_i)_{v'}$ and of $preconds^+(w)_{uv}$ with $preconds^+(w)_{v'}$. Similarly for case (b).

Parts (ii) and (iii) of the proof can be elaborated in the same way, but we will not belabor the matter further. Only in the correctness proof for n-ary disjunctive + sv-constraints (Theorem 12) will we again spell out some of the details involving the Unification Lemma.

Theorem 5 *The "dedicated" computation of sv-constraints in accordance with the hypothesize-and-test scheme is completed in polynomial time for any bound* max *on the allowable number of supplementary conditions, specifically in*

$$O\left(\sum_{o\in O} l_o^4 \left[\sum_{o'\in O} l_{o'}^3 \;+\; \frac{max}{l_o} \cdot \sum_{o'\in O} l_{o'}^{max+2} \;+\; n_{init}^2\right]\right)$$

steps, where O is the set of planning operators, l_o is the size of operator o, and n_{init} is the number of initial conditions.

Proof. The stated complexity bound is a simplification of

$$O\left(\sum_{o\in O} l_o^3 \left[\sum_{o'\in O} l_o l_{o'}^3 \;+\; \sum_{o'\in O} l_{o'} \left[l_{o'}(l_{o'} + l_o l_{o'} + l_{o'}) + l_o l_{o'}\right] + max \cdot \sum_{o'\in O} l_{o'}^{max+2} \;+\; l_o n_{init}^2\right]\right).$$

As in the proof of Theorem 3, the outer summation corresponds to the iteration over operators in the formation of hypotheses. The overall factor l_o^3 derives from the iteration over an operator's effects and concomitant changes (preconditions and effects) on which

hypothesis formation is based. The first inner sum corresponds to verification condition (2), and reflects the iteration over pairs of effects threatening single-valuedness (factor $l_{o'}^2$) and if appropriate the search for singleton excuses as per verification conditions (2a,b) (factor $l_o l_{o'}$). In the second inner sum, corresponding to verification condition (3), the overall multiplier $l_{o'}$ comes from the iteration over effects. The term $l_{o'}(l_{o'} + l_o l_{o'} + l_{o'})$ reflects (3a), with a multiplier $l_{o'}$ from the search for an appropriate precondition, and an embedded sum of three terms where the first, $l_{o'}$, corresponds to the search for a complementary effect (as per (3a(i))), and the second and third, $l_{o'} l_o + l_{o'}$, correspond to the search for a *when*-clause whose preconditions are entailed by a subset of the candidate supplementary conditions, and the search for a complementary effect in such a *when*-clause (as per (3a(ii))). The remaining term $l_o l_{o'}$ derives from the search for a singleton excuse (verification condition (3b)) when verification condition (3a) fails. The inner sum $max \cdot \sum_{o' \in O} {lo'}^{max+2}$ corresponds to the selection of irredundant sets of up to *max* supplementary conditions. (This relies on set-cover complexity, as in Theorem 3.) The final term $l_o n_{init}^2$ reflects the check of the hypothesis in the initial conditions; as discussed at the end of section 3.5, this could be improved to $l_o n_{init}$. □

Theorem 6 *Any implicative + sv-constraint*

```
((IMPLIES (P ?X ?*Y) (NOT (Q ?X))) (S1 ?X)...(Sk ?X))
```

(i.e., with the consequent variables subsumed by the antecedent variables) produced by DISCOPLAN *is correct.*

The notation `(Si ?X)` as before means that the ith supplementary condition may (but need not) involve `?X`. The theorem and its proof are easily generalized to the case where `P` and `Q` have several unstarred shared arguments, and `P` has several starred variables not occurring in the other. (Again, think of `?X, ?*Y` as two sets of arguments.)

Remark: We use `(S ?X)` for the conjunction of all supplementary conditions. Then the simultaneous constraints of the theorem, including supplementary conditions, state that in all reachable states,

(C1) $\forall x, y, y'.S(x) \wedge P(x, y) \wedge P(x, y') \Rightarrow y = y'$,

(C2) $\forall x, y.S(x) \wedge P(x, y) \Rightarrow \neg Q(x)$.

Proof. As in Theorems 3 and 5 the correctness proof is based on the sufficiency of the relevant verification conditions (as stated in subsection 4.3.1). We can take both constraints to be true in the initial state, since DISCOPLAN straightforwardly verifies them (verification condition 1). Given any instance of any operator o, and a prior state s_{prior} in which the preconditions of the primary *when*-clause w_1 of that operator instance are true, we assume for induction that the 2 constraints hold in s_{prior}. We want to show that they also hold in the resultant state s_{result}. Assume otherwise, i.e., assume that

(C3) for some constants a, b, c, where $b \neq c$, $S(a), P(a, b)$, and $P(a, c)$ hold in s_{result}; or

(C4) for some constants a, b, c, $S(a), P(a, b)$, and $\neg Q(a)$ hold in s_{result}.

We derive a contradiction for each of the two cases. First consider case (C3). Since S is static, we also had

(C5) $S(a)$ true in s_{prior}.

But by induction assumption (C1), at least one of the literals in (C3) must have been false in s_{prior}, and so

(C6) $P(a,b)$ or $P(a,c)$ was false in s_{prior}.

We now argue as follows. (A few details are added below.) (i) Since according to (C3) $P(a,b)$ and $P(a,c)$ are both true in s_{result} while according to (C6) one of them was false in s_{prior}, by the semantics of operators there must be an effect literal that was instantiated to $P(a,b)$ or $P(a,c)$. (ii) At least one of $P(a,b)$, $P(a,c)$ was true in s_{prior}, otherwise there would need to be multiple effects $P(a,b)$, $P(a,c)$ to render (C3) true, and this is ruled out by verification condition 2 (according to which a precondition of o would entail $\neg S(a)$, contrary to (C3)). (Details can be filled in as in subargument (i) in the proof of Theorem 5.) (iii) Since there was an effect $P(a,b)$ or $P(a,c)$ by (i), verification condition 3(a) or (b) applies. 3(b) entails that $\neg S(a)$ held in s_{prior}, contrary to assumption (C3); so 3(a) applies. Now by 3(a)(ii) we have a precondition $\neg Q(a)$ or a change from $P(a,d)$ to $\neg P(a,d)$ for some value of d, compensating for effect $P(a,b)$ or $P(a,c)$. If the former were true, $S(a) \wedge P(a,y)$ would be false for all values of y in s_{prior}, by the induction assumption. But $S(a)$ is true so $P(a,b)$ and $P(a,c)$ would be false in s_{prior}, contrary to (ii). So we are left with the possibility of a compensating change from $P(a,d)$ to $\neg P(a,d)$ for some $d \neq b$ or $\neq c$ depending on whether we have an effect $P(a,b)$ or $P(a,c)$. But if $P(a,d)$ held in s_{prior}, then by induction assumption (C1) $P(a,b)$ and $P(a,c)$ were false in s_{prior}, contrary to (ii).

Now consider case (C4). We begin much as before by noting that since S is static, (C5) holds, and that by (C5) and induction assumption (C2),

(C7) either $P(a,b)$ or $\neg Q(a)$ was false in s_{prior}.

We argue that there is an effect literal that was instantiated to $P(a,b)$ or $\neg Q(a)$ (by (C4) and (C7) and the semantics of operators). For the possibility that $P(a,b)$ was generated, verification condition 3(a) or 3(b) applies, but (b) implies $\neg S(a)$, which cannot be true by (C5); so 3(a) applies. By 3(a)(i) there should then be an effect or persistent precondition $Q(a)$, but this is contradicted by the assumption in (C4) that $\neg Q(a)$ holds in s_{result}. This eliminates the possibility that $P(a,b)$ was generated, so that

(C8) $P(a,b)$ was true in s_{prior},

and we are left with the possibility that $\neg Q(a)$ was generated, and $Q(a)$ was true in s_{prior}. This brings into play verification condition 4. Now 4(b) entails $\neg S(a)$, contrary to (C5), leaving us with 4(a), requiring a concomitant change from $P(a,d)$ to $\neg P(a,d)$ for some d. But then by (C8) and induction assumption (C1), either $S(a)$ was false in the prior state or $d = b$. The former is again ruled out by (C5); so with $d = b$, we have a concomitant change from $P(a,b)$ to $\neg P(a,b)$. But this contradicts (C4). □

Theorem 7 *Any exclusive state constraint*

```
((IMPLIES (P ?X ?*Y) (NOT (Q ?X ?*Z))) (S1 ?X)...(Sk ?X))
```

produced by DISCOPLAN *is correct.*

The notation `(Si ?X)` as before means that the ith supplementary condition may (but need not) involve `?X`. The theorem and its proof are easily generalized to the case where `P` and `Q` have several unstarred shared arguments, and each has several starred arguments not occurring in the other. (Again, think of `?X, ?*Y, ?*Z` as sets of arguments.)

Remark: We use `(S ?X)` for the conjunction of all supplementary conditions. Then the simultaneous constraints of the theorem, including supplementary conditions, state that in all reachable states,

(C1) $\forall x, y, y'.S(x) \wedge P(x, y) \wedge P(x, y') \Rightarrow y = y'$,

(C2) $\forall x, z, z'.S(x) \wedge Q(x, z) \wedge Q(x, z') \Rightarrow z = z'$, and

(C3) $\forall x, y, z.S(x) \wedge P(x, y) \Rightarrow \neg Q(x, z)$.

Proof. As in Theorems 3, 5 and 7 the correctness proof is based on the sufficiency of the relevant verification conditions (as stated in subsection 4.3.2). Much as in the previous proofs we can take all three constraints to be true in the initial state, since DISCOPLAN straightforwardly verifies them (verification condition 1). Given any instance of any operator o, and a prior state in which the preconditions of the primary *when*-clause w_1 of that operator instance are true, we assume for induction that the 3 constraints hold in that prior state. We want to show that they also hold in the resultant state. Assume otherwise, i.e., assume that

(C4) for some constants a, b, c, where $b \neq c$, $S(a), P(a, b)$, and $P(a, c)$ hold in the resultant state; or

(C5) for some constants a, b, c, where $b \neq c$, $S(a), Q(a, b)$, and $Q(a, c)$ hold in the resultant state; or

(C6) for some constants a, b, c, $S(a), P(a, b)$, and $Q(a, c)$ hold in the resultant state.

We derive a contradiction for each of the three cases. First consider case (C4). Since S is static, we also had

(C7) $S(a)$ true in the prior state.

But by the induction assumption, at least one of the literals in (C4) must have been false in the prior state, and so

(C8) $P(a, b)$ or $P(a, c)$ was false in the prior state.

We now argue, much as in the proof of Theorem 7, that (i) there is an effect literal that was instantiated to $P(a, b)$ or $P(a, c)$; (ii) at least one of $P(a, b)$, $P(a, c)$ was true in the prior state; (iii) by (i), and the way hypotheses are tested and augmented with supplementary conditions (verification condition 4a,b), $Q(a, d)$ was true for some d in the prior state; (iv) so, by the induction assumption (C3) and by (C7), $P(a, y)$ was false for all y in the prior state, contrary to (ii).

The details for steps (i) and (ii) of the argument are just as in the proof of Theorem 7, and we will not repeat them. Proceeding to (iii), we note that in testing the exclusion hypothesis under consideration, we ensure that whenever an effect of form $P(x, y)$ is present, a precondition of form $Q(x', z)$ is present in the corresponding *when*-clause or primary *when*-clause with x, x' provably equal (verification condition 4a), or else we introduce a supplementary condition that is false whenever the preconditions of effect $P(x, y)$ hold (verification conditions 4b). But by (i) such an effect is indeed present, namely $P(r, s)$, so either there is a precondition $Q(r', z)$ of w or w_1, with r, r' provably equal, or else $S(r)$ is false whenever the preconditions of w and w_1 hold. The latter is ruled out by (C7) (the truth of $S(a)$) and the presumed truth of the preconditions of w and w_1 in the prior state. Thus $Q(a, d)$ holds for some d in the prior state. But then from induction assumption (C3) in the prior state, we have $\forall y.\neg S(a) \vee \neg P(a, y)$ in the prior state; in particular, $\neg S(a) \vee \neg P(a, b)$ and $\neg S(a) \vee \neg P(a, c)$, but at least one of these is false by (C7) and (ii).

The derivation of a contradiction from (C5) is completely analogous to the previous argument for (C4) (with P and Q interchanged), so we proceed to (C6). We begin as before by noting that (C7) holds, and that by (C7) and the induction assumption,

(C9) either $P(a, b)$ or $Q(a, c)$ was false in the prior state.

We argue that (i) there is an effect literal that was instantiated to $P(a, b)$ or $Q(a, c)$; for the case where $P(a, b)$ was generated, we argue further that (ii) $Q(a, d)$ was verified as a precondition and $\neg Q(a, d)$ was generated as an effect for some d, and (iii) consequently $\neg Q(a, z)$ holds for all z in the resultant state, contrary to (C6); and (iv) an analogous argument holds for the case where $Q(a, c)$ was generated.

(i) is immediate from (C6) and (C9). So assume that the effect $P(a, b)$ was generated by an effect literal $P(r, s)$ of some *when*-clause w (possibly w_1). Concerning (ii), in virtue of verification conditions (4a,b), either there exists a precondition $Q(r', z)$ of w or w_1 where r and r' are provably identical, or else there is a supplementary condition among the conjuncts in $S(r)$ whose falsity is entailed by the preconditions of w or w_1. The latter is impossible by the presumed truth of the preconditions of w and w_1 and by (C7) for the operator instance under consideration. So precondition $Q(a, d)$ is verified in the prior state. Verification conditions (4a,b) further guarantee that either there is an effect $\neg Q(r'', z')$ of w or w_1, where r', r'' are provably equal and z, z' are provably equal, or else there is such an effect of some other *when*-clause w_2, where the supplementary conditions of the hypothesis entail the truth of the preconditions of w_2. So in view of (C7), we do indeed have an effect $\neg Q(a, d)$ for the operator instance in question. Concerning (iii), by induction assumption (C2) in the prior state, by (C7), and by the truth of $Q(a, d)$ in the prior state we have $\forall z.Q(a, z) \Rightarrow z = d$ in the prior state. But since we have an effect $\neg Q(a, d)$, we will have $\forall z.\neg Q(a, z)$ in the resultant state, unless there is some additional effect of form $Q(u, v)$ of some *when*-clause w_3 (not necessarily distinct from w, w_1, or w_2) with true preconditions and with u instantiated to a. But this latter possibility is ruled out by verification condition (6), i.e., if there were such an effect (given that there is an effect $P(r, s)$) there would be a supplementary condition whose falsity is entailed by the preconditions of w, w_1, or w_3. So since $\forall z.\neg Q(a, z)$ contradicts (C6) in the resultant state, we have established (iii). Finally the analogous argument for (iv) essentially just interchanges the roles of P and Q and of b and c. □

Theorem 8 *The computation of exclusive state constraints in accordance with the hypothesize-and-test scheme is completed in polynomial time for any bound* max *on the allowable number of supplementary conditions.*

Proof sketch. We omit details. The analysis here is very much as in Theorems 3 and 5. Verification conditions (1-6) for exclusive state constraints closely resemble those for simple implicative constraints and for (independently verifiable) sv-constraints, and lead to similar time bounds, polynomial in the sizes l_o of operators and the number n_{init} of initial conditions, for any fixed upper bound *max* on the number of supplementary conditions. □

References

[1] V. Alcázari and A. Torralba. A reminder about the importance of computing and exploiting invariants in planning. In *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling*, pages 2–6, 2015.

[2] J. Allen, L. Schubert, G. Ferguson, P. Heeman, C. Hwang, T. Kato, M. Light, N. Martin, B. Miller, M. Poesio, and D. Traum. The TRAINS project: A case study in defining a conversational planning agent. *Journal of Experimental and Theoretical AI*, 7:7–48, 1995.

[3] M. Asai and A. Fukunaga. Fully automated cyclic planning for large-scale manufacturing domains. In *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling*, 2014.

[4] B.C. B. Cenk Gazen and C.A. Craig A. Knoblock. Combining the expressiveness of UCPOP with the efficiency of Graphplan. In S. Steel and R. Alami, editors, *Recent Advances in AI: Planning: 4th European Conference on Planning, ECP-97*. Springer-Verlag, 1997.

[5] F. Bacchus and C.B. Fraser. Inner and outer boundaries of literals: A mechanism for computing domain specific information. In *Working Notes of the AIPS00 Workshop on Analysing and Exploiting Domain Knowledge for Efficient Planning*, pages 29–36. AAAI press, 2000.

[6] C. Bačkstroḿ and B. Nebel. Complexity results for SAS+ planning *Computational Intelligence*, 11(4):625–655, 1995.

[7] S. Bernardini, F. Fagnani, and D.E. Smith. Extracting mutual exclusion invariants from lifted temporal planning domains. *Artificial Intelligence*, 258:1–65, 2018.

[8] B. Bonet and H. Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1–2), 2001.

[9] Y. Chen, R. Huang, Xingm Z., and W. Zhang. Long-distance mutual exclusion for planning. *Artificial Intelligence*, 173(2):365–391, 2009.

[10] M.C. Cooper, M. de Roquemaurel, and P. Régnieri. A weighted CSP approach to cost-optimal planning. *AI Communumication*, 24(1):1–29, 2011.

[11] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[12] C. Domshlak, J. Hoffmann, and M. Katz. Red-black planning: A new systematic approach to partial delete relaxation. *Artificial Intelligence*, 221:73–114, 2015.

[13] S. Edelkamp and M. Helmert. Exhibiting knowledge in planning problems to minimize state encoding length. In *Recent Advances in AI Planning. 5th European Conference on Planning (ECP-99)*, volume 1809 of *Lecture Notes in Artificial Intelligence*, pages 135–147. Springer-Verlag, 1999.

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

[15] Michael Genesereth and Nils J. Nilsson. *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1987.

[16] A. Gerevini and Schubert. L. Discovering state constraints for planning: DISCOPLAN. Technical Report 2005-09-48, Dip. di Elettronica per l'Automazione, Università degli Studi di Brescia, 2005. Revised version of Technical Report 811, Dept. of Computer Science, University of Rochester, 2003.

[17] A. Gerevini, A. Saetti, and I. Serina. Planning through stochastic local search and temporal action graphs. *Journal of Artificial Intelligence Research (JAIR)*, 20:239–290, 2003.

[18] A. Gerevini and L. Schubert. Accelerating Partial-Order Planners: Some Techniques for Effective Search Control and Pruning. *Journal of Artificial Intelligence Research (JAIR)*, 5:95–137, Sept. 1996.

[19] 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.

[20] A. Gerevini and L. Schubert. Extending the types of state constraints discovered by DISCOPLAN. In *Working Notes of the AIPS00 Workshop on Analysing and Exploiting Domain Knowledge for Efficient Planning*, pages 4–11, 2000.

[21] 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.

[22] A. Gerevini and L.K. Schubert. DISCOPLAN: an efficient on-line system for computing planning domain invariants. In *Proceedings of the European Conference on Planning (ECP-01)*, pages 284–288. AAAI Press, 2014.

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

[24] J. Haslum and U. Scholz. Domain Knowledge in Planning: Representation and Use. In *Proceedings of the 13th International Conference on Automated Planning & Scheduling (ICAPS-03), Workshop on PDDL*, pages 69–78. 2003.

[25] M. Helmert. Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence*, 173(5-6):503–535, 2009.

[26] G. Kelleher. Determining general consequences of sets of actions. Technical Report TR CMS.14.96, Liverpool Moores University, 1996.

[27] J. Kelleher and A. Cohn. Automatically synthesising domain constraints from operator descriptions. In *Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI-92)*, pages 653–655. Wiley, 1992.

[28] J. Kvarnström. Applying domain analysis techniques for domain-dependent control in Talplanner. In *Proceedings of the Sixth International Conference on Artificial Intelligence Planning Systems*, pages 101–111, 2002.

[29] F. Lin. Discovering state invariants. In *Proceedings of the Ninth International Conference on Principles of Knowledge Representation and Reasoning (KR-04)*, pages 536–544. AAAI Press, 2004.

[30] N. Lipovetzkyi, C.J. Muise, and G. Geffner. Traps, invariants, and dead-ends. In *Proceedings of the Twenty-Sixth International Conference on Automated Planning and Scheduling, ICAPS 2016.*, pages 211–215, 2016.

[31] D. McAllester. Observations on cognitive judgements. In *Proceedings of the Nineth National Conference of the American Association for Artificial Intelligence (AAAI-91)*, pages 910–914. MIT Press, 1991.

[32] P. Morris and R. Feldman. Automatically derived heuristics for planning search. In *Proceedings of the Second Irish Conference on AI and Cognitive Science*. School of Computer Science Applications, Dublin City University, 1989.

[33] B. Nebel. On the compilability and the expressive power of propositional planning formalisms. *Journal of Artificial Intelligence Research (JAIR)*, 12:271–315, 2000.

[34] 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.

[35] M.F. Rankoohi and G. Ghassem-Sani. ITSAT: an efficient sat-based temporal planner. *Journal of Artificial Intellsigence Research*, 53:541–632, 2015.

[36] J. Rintanen. A planning algorithm not based on directional search. In *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR-98)*, pages 617–624. Morgan Kaufmann, 1998.

[37] J. Rintanen. An iterative algorithm for synthesizing invariants. In *Proceedings of the Seventeenth National Conference of the American Association for Artificial Intelligence (AAAI-00)*, pages 806–811. AAAI press / The MIT Press, 2000.

[38] J. Rintanen. Regression for classical and nondeterministic planning. In *ECAI 2008 - 18th European Conference on Artificial Intelligence, Proceedings*, pages 568–572, 2008.

[39] J. Rintanen. Constraint-based algorithm for computing temporal invariants. In *Logics in Artificial Intelligence - 14th European Conference, Proceedings*, pages 665–673, 2014.

[40] J. Rintanen. Schematic invariants by reduction to ground invariants. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence,*, pages 3644–3650, 2017.

[41] U. Scholz. Extracting state constraints from PDDL-like planning domains. In *Working Notes of the AIPS00 Workshop on Analyzing and Exploiting Domain Knowledge for Efficient Planning*, pages 43–48. AAAI press, 2000.