Planning as satisfiability: parallel plans and algorithms for plan search

Jussi Rintanen
National ICT Australia
Canberra Research Laboratory
Australia

Keijo Heljanko and Ilkka Niemelä
Helsinki University of Technology
Laboratory for Theoretical Computer Science
P. O. Box 5400, FI-02015 TKK
Finland

We address two aspects of constructing plans efficiently by means of satisfiability testing: efficient encoding of the problem of existence of plans of a given number t of time points in the propositional logic and strategies for finding plans, given these formulae for different values of t.

For the first problem we consider three semantics for plans with parallel operator application in order to make the search for plans more efficient. The standard semantics requires that parallel operators are independent and can therefore be executed in any order. We consider a more relaxed definition of parallel plans which was first proposed by Dimopoulos et al., as well as a normal form for parallel plans that requires every operator to be executed as early as possible. We formalize the semantics of parallel plans emerging in this setting and present translations of these semantics into the propositional logic. The sizes of the translations are asymptotically optimal. Each of the semantics is constructed in such a way that there is a plan following the semantics exactly when there is a sequential plan, and moreover, the existence of a parallel plan implies the existence of a sequential plan with as many operators as in the parallel one.

For the second problem we consider strategies based on testing the satisfiability of several formulae representing plans of n time steps for several values of n concurrently by several processes. We show that big efficiency gains can be obtained in comparison to the standard strategy of sequentially testing the satisfiability of formulae for an increasing number of time steps.

Contents

1. INTRODUCTION

In the simplest form of planning, traditionally called classical planning, the objective is to find a sequence of actions that leads from a given initial state to one of the goal states. Planning as satisfiability [Kautz and Selman 1996] is a leading approach to solve this kind of planning problems. The underlying idea is to encode the bounded plan existence problem, i.e. whether a plan of a given length n exists, as a formula in the classical propositional logic. The formula for a given n is satisfiable if and only if there is a plan of length n. Finding a plan reduces to testing the satisfiability of the formulae for different values of n.

An important factor in the efficiency of planning as satisfiability in finding non-optimal plans which do not necessarily have the smallest possible number of operators is the notion of parallel plans [Blum and Furst 1997; Kautz and Selman 1996]. In a parallel plan each time point may have more than one operator. The length parameter n restricts the number of time points but not directly the number of operators. A parallel plan is a representation of one or more sequential plans, and the parallelism is not meant to represent genuine temporal parallelism. Different notions of parallel plans can be defined, and for maintaining the connection to sequential plans it is required that there is a parallel plan exactly when there is a sequential plan, and moreover, mapping a given parallel plan to a sequential plan should be possible in polynomial time. In this paper we develop semantics for parallel plans that have these properties. Results in Section 2 identify important tractable and intractable notions of parallel plans.

Parallel plans increase the efficiency of planning for two reasons. First, since several independent operators can be parallel at one time point, it is unnecessary to consider all their total orderings during plan search, unlike with sequential plans. Second, increased parallelism leads to a decreased number of time points. This reduces the number of propositional variables and the size of formulae and makes satisfiability testing more efficient.

The standard *state-based encoding* of parallel plans [Kautz and Selman 1996] allows several operators at the same time point as long as the operators are mutually non-interfering. This condition guarantees that any total ordering on the simultaneous operators is a valid execution and that it leads to the same state in all cases. We formalize a generalization of this idea and call it *the* $\forall$*-step semantics* of parallel plans, and give asymptotically optimal linear-size encodings of this semantics in the classical propositional logic.

Our objective is to develop more efficient techniques for different forms of planning, and for this purpose we formalize further two semantics of parallel plans and present efficient encodings of them in the propositional logic. Both of these semantics are known from earlier research but the first, *the process semantics*, has not been considered in connection with planning, and the second, *the* $\exists$*-step semantics*, has not been given efficient encodings in the propositional logic before.

The two new semantics considered in this paper are orthogonal refinements of the $\forall$-step semantics. The process semantics is stricter than the $\forall$-step semantics in that it requires all actions to be taken as early as possible. Since there are less valid plans of a given length according to the process semantics than the $\forall$-step semantics, the corresponding satisfiability problems are more strongly constrained and plan search could be more efficient. Process semantics was first introduced for Petri nets; for an overview see [Best and Devillers 1987]. Heljanko [2001] has applied this semantics to the deadlock detection of 1-safe Petri nets and has demonstrated that it leads to big efficiency gains for many types of problems.

The idea of the $\exists$-step semantics was proposed by Dimopoulos et al. [1997]. They pointed out that it is not necessary that all parallel operators are non-interfering as long as they can be executed in at least one order, which makes it possible to increase the number of parallel operators still further. They also showed how certain planning problems can be modified to satisfy this condition and that the reduction in the number of time points improves runtimes. Until now the application of $\exists$-step semantics in planning as satisfiability was hampered by the cubic size of the obvious encodings. We give more compact encodings for this semantics and show that this often leads to dramatic improvements in efficiency. Before the developments reported in this paper, this semantics was never used in an automated planner that is based on a declarative language like the propositional logic. Our most efficient encodings of this and the other semantics are more restrictive than the general definitions of these semantics. We justify the restrictions by showing that the general definitions are intractable.

As a second contribution of this paper we demonstrate the strong potential of the planning as satisfiability approach to solve non-optimal planning efficiently by proposing two new concurrent algorithms for controlling any planner that is based on testing the existence of plans with different numbers of time points. These algorithms effectively avoid the expensive plan inexistence (unsatisfiability) tests that dominate the runtimes of earlier planners. The speed-up of these algorithms over the standard sequential algorithm can be arbitrarily high, and these algorithms are guaranteed to be only at most a constant factor slower than the standard sequential algorithm. An empirical investigation (Section 5) shows that for some classes of problems for which planning as satisfiability had not earlier fared very well these algorithms lift the efficiency to a completely different level.

The results of this paper are also directly applicable to bounded model checking [Biere, Cimatti, Clarke, and Zhu 1999] of safety properties in computer-aided verification. It is also possible to extend the results to model checking for arbitrary linear temporal logic (LTL) properties. We do not pursue this topic further in this paper.

The structure of this paper is as follows. In Sections 2.1, 2.2, and 2.3 we define the standard $\forall$-step semantics, the process semantics, and the $\exists$-step semantics, respectively. A main result of Section 2 is the identification of the border between tractable and intractable notions of parallel plans, based on the distinction between polynomial-time and NP-hard decision problems.

In Section 3 we present encodings of classical planning under the different semantics of parallel plans in the classical propositional logic. Section 3.1 presents the part of the encodings shared by all the semantics, and Sections 3.2, 3.3 and 3.4 presents encodings of the three semantics. A main result is the introduction of encodings that have a size that is asymptotically optimal. Encodings of planning with this property have not been presented earlier.

Section 4 evaluates the efficiency of the different semantics for different kinds of planning problems. Section 4.3 makes a comparison in terms of runtimes and plan quality with difficult problems which are sampled from the space of all problem instances. Section 4.4 makes a comparison with a number of standard benchmark problems.

Section 5 presents two new concurrent algorithms for controlling any planner that uses as the main subprocedure a test for the existence of plans with a given number of time points. In Section 5.4 the properties of the algorithms are analytically investigated, and in Section 5.5 their impact on planner runtimes is experimentally demonstrated.

Section 6 discusses related work and Section 7 concludes the paper.

1.1 Notation

We consider planning in a setting where the states of the world are represented in terms of a set A of Boolean state variables which take the value *true* or *false*. Formulae are formed from the state variables with the connectives $\vee$, $\wedge$ and $\neg$. The connectives $\rightarrow$ and $\leftrightarrow$ are defined in terms of the other connectives. Each *state* is a valuation of A, which is an assignment $s : A \rightarrow \{0, 1\}$. A *literal* is a formula of the form a or $\neg a$ where $a \in A$ is a state variable. We define the *complements* of literals as $\overline{a} = \neg a$ and $\overline{\neg a} = a$ for all $a \in A$. A *clause* is a disjunction $l_1 \vee \cdots \vee l_n$ of one or more literals. We also use the constant atoms $\top$ and $\bot$ for denoting *true* and *false*, respectively.

We use *operators* for expressing how the state of the world can be changed.

Definition 1.1 *An* operator *on a set of state variables A is a triple $\langle p, e, c \rangle$ where*

(1) *p is a propositional formula on A (*the precondition*),*
(2) *e is a set of literals on A (*the unconditional effects*), and*
(3) *c is a set of pairs $f \rhd d$ (*the conditional effects*) where f is a propositional formula on A, and d is a set of literals on A.*

For an operator $\langle p, e, c\rangle$ its *active effects* in state s are

$$[o]_s = e \cup \bigcup\{d|f \rhd d \in c, s \models f\}.$$

The operator is *executable* in s if $s \models p$ and its set of active effects in s is consistent (does not contain both a and $\neg a$ for any $a \in A$.) If this is the case, then we define $\mathit{app}_o(s)$ as the unique state that is obtained from s by making $[o]_s$ true and retaining the values of the state variables not occurring in $[o]_s$. For sequences $o_1; o_2; \ldots; o_n$ of operators we define $\mathit{app}_{o_1;o_2;\ldots;o_n}(s)$ as $\mathit{app}_{o_n}(\cdots \mathit{app}_{o_2}(\mathit{app}_{o_1}(s))\cdots)$. For sets S of operators and states s we define $\mathit{app}_S(s)$: the result of simultaneously applying all operators $o \in S$. We require that $\mathit{app}_o(s)$ is defined for every $o \in S$ and that the set $[S]_s = \bigcup_{o\in S}[o]_s$ of active effects of all operators in S is consistent. For operators $o = \langle p, e, c\rangle$ and atomic effects l of the form a and $\neg a$ (for $a \in A$) define the *effect precondition* $\mathit{EPC}_l(o) = \top$ if $l \in e$ and otherwise $\mathit{EPC}_l(o) = \bigvee\{f|f \rhd d \in c, l \in d\}$ where the empty disjunction $\bigvee \emptyset$ is defined as $\perp$.

Lemma 1.2 *For literals l, operators o and states s, $l \in [o]_s$ if and only if $s \models \mathit{EPC}_l(o)$.*

We sometimes consider operators without conditional effects and disjunctivity in preconditions: $\langle p, e, c\rangle$ is a *STRIPS operator* if $c = \emptyset$ and p is a conjunction of literals. Let $\pi = \langle A, I, O, G\rangle$ be a *problem instance* consisting of a set A of state variables, a state I on A (the initial state), a set O of operators on A, and a formula G on A (the goal formula). A (sequential) *plan* for π is a sequence $\sigma = o_1; \ldots; o_n$ of operators from O such that $\mathit{app}_\sigma(I) \models G$. This means that applying the operators in the given order starting in the initial state is defined (the precondition of every operator is true and the active effects are consistent when the operator is applied) and produces a state that satisfies the goal formula. Sometimes we say that an operator sequence is a plan for O and I when we simply want to say that the plan is executable starting from I without specifying the goal states.

In the rest of this paper we also consider plans that are sequences of *sets of operators*. The different semantics discussed in the next sections impose further constraints on these sets.

2. DEFINITIONS OF PARALLEL PLANS

2.1 $\forall$-Step semantics

We formally present a semantics that generalizes the semantics used in most works on parallel plans, for example that of Kautz and Selman [1996].

Earlier definitions of parallel plans have been based on the notion of *interference*. The parallel application of a set of operators is possible if the operators do not interfere. Lack of interference guarantees that the operators can be executed sequentially in any total order and that the terminal state is independent of the ordering. As shown in Theorem 2.3, non-interference and executability in any order coincide for STRIPS operators. Our definition of operators extends the definition of STRIPS operators considerably, and instead of non-interference in Definition 2.1 we adopt the more abstract and intuitive order-independence as the basic principle in the $\forall$-step semantics.

For the efficiency of plan search and plan validation it is important that the test whether a plan is executable and achieves the goals is tractable. For this reason we investigate the tractability of our general definition of $\forall$-step semantics and then identify restricted tractable classes of $\forall$-step plans. This investigation goes beyond earlier works like by Blum and Furst [1997] and Kautz and Selman [1996, 1999] which restrict to STRIPS operators.

Definition 2.1 ($\forall$-Step plans) *For a set of operators O and an initial state I,* a $\forall$-step plan for O and I *is a sequence $T = \langle S_0, \dots, S_{l-1} \rangle$ of sets of operators for some $l \geq 0$ such that there is a sequence of states $s_0, \dots, s_l$ (the execution of T) such that*

(1) *$s_0 = I$, and*

(2) *for all $i \in \{0, \dots, l-1\}$ and every total ordering $o_1, \dots, o_n$ of S_i, $app_{o_1;\dots;o_n}(s_i)$ is defined and equals s_{i+1}.*

We show that this abstract definition yields the standard definition of parallel plans for STRIPS operators which requires that no operator falsifies the precondition of any other operator that is applied simultaneously.

Lemma 2.2 *Let $T = \langle S_0, \dots, S_{l-1} \rangle$ be a $\forall$-step plan with execution $s_0, \dots, s_l$. Then the following hold.*

(1) *There is no $i \in \{0, \dots, l-1\}$ and $\{\langle p, e, c \rangle, \langle p', e', c' \rangle\} \subseteq S_i$ and $a \in A$ such that $a \in e$ and $\neg a \in e'$.*

(2) *$app_o(s_i)$ is defined for every $o \in S_i$.*

PROOF. For (1) we derive a contradiction by assuming the opposite. Take an ordering of the operators such that $\langle p, e, c \rangle$ and $\langle p', e', c' \rangle$ are the last operators in this order. Hence $s_{i+1} \models \neg a$. But the ordering in which the two operators are the other way round leads to a state s'_{i+1} such that $s'_{i+1} \models a$. This contradicts the assumption that T is a $\forall$-step plan. Hence (1) holds.

Consider any operator $o \in S_i$ and any ordering in which o is the first operator. For the operators to be executable in this order, o has to be executable in s_i. Therefore (2). $\square$

For operators without conditional effects (including STRIPS operators) the above lemma means that for every set S_i of parallel operators $app_{S_i}(s_i)$ is defined. With conditional effects sequential execution in any order is sometimes possible even when simultaneous execution is not: consider for example $\{\langle \top, \emptyset, \{(\neg a \wedge \neg b) \rhd \{a, \neg b\}, b \rhd \{a\}\} \rangle, \langle \top, \emptyset, \{(\neg a \wedge \neg b) \rhd \{\neg a, b\}, a \rhd \{b\}\} \rangle\}$ executed in a state that satisfies $\neg a \wedge \neg b$.

Theorem 2.3 *Let O be a set of STRIPS operators, I a state, and $T = \langle S_0, \dots, S_{l-1} \rangle \in (2^O)^l$. Then T is a $\forall$-step plan for O and I if and only if there is a sequence of states $s_0, \dots, s_l$ such that*

(1) *$s_0 = I$,*

(2) *$s_{i+1} = app_{S_i}(s_i)$ for all $i \in \{0, \dots, l-1\}$, and*

(3) *for no $i \in \{0, \dots, l-1\}$ and two operators $\{\langle p, e, \emptyset \rangle, \langle p', e', \emptyset \rangle\} \subseteq S_i$ there is $m \in e$ such that $\overline{m}$ is one of the conjuncts of p'.*

PROOF. We first prove the *only if* part. Since T is a $\forall$-step plan, it has an execution $s_0, \dots, s_l$ as in Definition 2.1. We show that the three conditions on the right side of the equivalence are satisfied by this sequence of states.

By the definition of $\forall$-step plans, the first state of the execution is the initial state I. Hence we get (1).

By (1) of Lemma 2.2 for all $i \in \{0, \dots, l-1\}$ the sets $E_i = [S_i]_{s_i} = \bigcup\{e|\langle p, e, \emptyset\rangle \in S_i\}$ are consistent. By (2) of the same lemma the preconditions of all operators in S_i are true in s_i. Hence the state $\mathit{app}_{S_i}(s_i)$ is defined. The changes made by any total ordering of S_i equal E_i because the effects of no operator in S_i override any effect of another operator in S_i. Therefore $s_{i+1} = \mathit{app}_{S_i}(s_i)$. This establishes (2).

For the sake of argument assume that there is literal m and $i \in \{0, \dots, l-1\}$ so that $m \in e$ for some $\langle p, e, \emptyset\rangle \in S_i$ and $\overline{m}$ is a conjunct of the precondition p' of some other $\langle p', e', \emptyset\rangle \in S_i$. Then in every total ordering of the operators in which $\langle p, e, \emptyset\rangle$ immediately precedes $\langle p', e', \emptyset\rangle$ the latter would not be executable. This, however, contradicts the definition of $\forall$-step plans. Therefore (3).

Then we prove the *if* part. Assume there is a sequence $s_0, \dots, s_l$ satisfying (1), (2) and (3). We show that T and $s_0, \dots, s_l$ satisfy Definition 2.1 of $\forall$-step plans.

That $s_0 = I$ is directly by our assumption (1).

We show that $\mathit{app}_{o_1;\dots;o_n}(s_i) = \mathit{app}_{S_i}(s_i)$ for all $i \in \{0, \dots, l-1\}$ and all total orderings $o_1, \dots, o_n$ of S_i. Since $\mathit{app}_{S_i}(s_i)$ is defined, the precondition of every $o \in S_i$ is true in s_i and $E_i = \bigcup\{e|\langle p, e, \emptyset\rangle \in S_i\}$ is consistent. Take any total ordering $o_1, \dots, o_n$ of S_i. By (3) no operator in S_i can disable another operator in S_i. Hence $\mathit{app}_{o_1;\dots;o_n}(s_i)$ is defined. Since E_i is consistent, effects of no operator can be overridden by another operator in S_i. Hence $\mathit{app}_{S_i}(s_i) = s_{i+1} = \mathit{app}_{o_1;\dots;o_n}(s_i)$. Since this holds for any total ordering of S_i, the definition of $\forall$-step plans is fulfilled. $\square$

Testing whether a sequence of sets of STRIPS operators is a $\forall$-step plan can be done in polynomial time. A simple quadratic algorithm tests the operators pairwise for occurrences of a literal and its complement in the effects of the two operators and in the effect of one and in the precondition of the other. Computing the successor states is similarly polynomial time computation.

In the general case, however, the definition of $\forall$-step plans is computationally rather complex. The next theorem gives the justification for restricting to a narrow class of $\forall$-step plans in the following. The proof of the theorem shows that co-NP-hardness holds even when operators have no conditional effects. Hence the high complexity emerges merely from disjunctivity in operator preconditions.

Theorem 2.4 *Testing whether a sequence of sets of operators is a $\forall$-step plan is co-NP-hard.*

PROOF. The proof is by reduction from TAUT. Let ϕ be any propositional formula. Let $A = \{a_1, \dots, a_n\}$ be the set of propositional variables occurring in ϕ. The set of state variables is A. Let $o_z = \langle \phi, \emptyset, \emptyset\rangle$. Let $S = \{\langle \top, \{a_1\}, \emptyset\rangle, \dots, \langle \top, \{a_n\}, \emptyset\rangle, o_z\}$. Let s and s' be states such that $s \not\models a$ and $s' \models a$ for all $a \in A$. We show that ϕ is a tautology if and only if $T = \langle S\rangle$ is a $\forall$-step plan for S and s.

Assume ϕ is a tautology. Now for any total ordering $o_0, \dots, o_n$ of S the state $\mathit{app}_{o_0;\dots,o_n}(s)$ is defined and equals s' because all preconditions are true in all states, and the set of effects of all operators is A (the set is consistent and making the effects true in s yields s'.) Hence T is a $\forall$-step plan.

Assume T is a $\forall$-step plan. Let v be any valuation. We show that $v \models \phi$. Let $S_v = \{\langle \top, \{a\}, \emptyset \rangle | a \in A, v \models a\}$. The operators S can be ordered to $o_0, \ldots, o_n$ so that the operators $S_v = \{o_0, \ldots, o_k\}$ precede o_z and $S \backslash (S_v \cup \{o_z\})$ follow o_z. Since T is a $\forall$-step plan, $app_{o_0;\ldots;o_n}(s)$ is defined. Since also $app_{o_0;\ldots;o_k;o_z}(s)$ is defined, the precondition ϕ of o_z is true in $v = app_{o_0;\ldots;o_k}(s)$. Hence $v \models \phi$. Since this holds for any valuation v, ϕ is a tautology. $\square$

Membership in co-NP is easy to show. There is a nondeterministic polynomial-time algorithm that can determine that a sequence of sets of operators is not a $\forall$-step plan. It first guesses an index i and a total ordering for the first $i-1$ steps and two total orderings for step i and then computes the two states that are reached by applying the operators in the first $i-1$ steps followed by one total ordering of step i. If the states differ or if not all operators are executable, then the definition of $\forall$-step plans is not fulfilled.

To obtain a tractable notion of $\forall$-step plans for all operators we can generalize the notion of interference used for STRIPS operators to arbitrary operators. Lack of interference is a sufficient but not necessary condition for a set of operators to be executable in every order with the same results. First we define positive and negative occurrences of state variables $a \in A$ in a formula inductively as follows.

Definition 2.5 (Positive and negative occurrences) *We say that a state variable a* occurs positively in ϕ *if positive(a, ϕ) is true. Similarly, a* occurs negatively in ϕ *if negative(a, ϕ) is true.*

$$\begin{aligned}
\textit{positive}(a, a) &= \textit{true, for all } a \in A \\
\textit{positive}(a, b) &= \textit{false, for all } \{a, b\} \subseteq A \textit{ such that } a \neq b \\
\textit{positive}(a, \phi \wedge \phi') &= \textit{positive}(a, \phi) \textit{ or positive}(a, \phi') \\
\textit{positive}(a, \phi \vee \phi') &= \textit{positive}(a, \phi) \textit{ or positive}(a, \phi') \\
\textit{positive}(a, \neg\phi) &= \textit{negative}(a, \phi)
\end{aligned}$$

$$\begin{aligned}
\textit{negative}(a, b) &= \textit{false, for all } \{a, b\} \subseteq A \\
\textit{negative}(a, \phi \wedge \phi') &= \textit{negative}(a, \phi) \textit{ or negative}(a, \phi') \\
\textit{negative}(a, \phi \vee \phi') &= \textit{negative}(a, \phi) \textit{ or negative}(a, \phi') \\
\textit{negative}(a, \neg\phi) &= \textit{positive}(a, \phi)
\end{aligned}$$

A state variable a occurs in ϕ *if it occurs positively or occurs negatively in ϕ.*

Below we also consider positive and negative occurrences of state variables in effects. A state variable a *occurs positively as an effect* in operator $\langle p, e, c \rangle$ if $a \in e$ or if there is $f \rhd d \in c$ so that $a \in d$. A state variable a *occurs negatively as an effect* in operator $\langle p, e, c \rangle$ if $\neg a \in e$ or there is $f \rhd d \in c$ such that $\neg a \in d$.

Definition 2.6 (Interference) *Let A be a set of state variables. Operators $o = \langle p, e, c \rangle$ and $o' = \langle p', e', c' \rangle$ over A* interfere *if there is $a \in A$ that*

(1) *occurs positively as an effect in o and occurs in f for some $f \rhd d \in c'$ or occurs negatively in p',*

(2) *occurs positively as an effect in o' and occurs in f for some $f \rhd d \in c$ or occurs negatively in p,*

(3) *occurs negatively as an effect in o and occurs in f for some $f \rhd d \in c'$ or occurs positively in p', or*

(4) *occurs negatively as an effect in o' and occurs in f for some $f \rhd d \in c$ or occurs positively in p.*

Proposition 2.7 *Testing whether two operators interfere can be done in polynomial time in the size of the operators.*

There are simple examples of valid $\forall$-step plans in which operators interfere according to the above definition. Hence the restriction to steps without interfering operators rules out many plans covered by the general definition (Definition 2.1.)

Example 2.8 Consider a set A of state variables and any set S of operators of the form

$$\langle \top, \emptyset, \{a \rhd \{\neg a\} | a \in A'\} \cup \{\neg a \rhd \{a\} | a \in A'\}\rangle$$

where A' is any subset of A (dependent on the operator.) Hence each operator reverses the values of a certain set of state variables. Executing the operators in any order results in the same state in every case. Hence $\langle S \rangle$ is a $\forall$-step plan according to Definition 2.1 but any two operators affecting the same state variable interfere. ■

Before formally connecting the notion of interference to plans that satisfy the $\forall$-step semantics we define a more relaxed notion of interference that is dependent on the state. In Section 3 we primarily use the state-independent notion of interference.

Definition 2.9 (Interference in a state) *Let A be a set of state variables. Operators $o = \langle p, e, c\rangle$ and $o' = \langle p', e', c'\rangle$ over A* interfere in a state *s if there is $a \in A$ so that*

(1) *$a \in [o]_s$ and a occurs in d for some $d \rhd f \in c'$ or occurs negatively in p',*
(2) *$a \in [o']_s$ and a occurs in d for some $d \rhd f \in c$ or occurs negatively in p,*
(3) *$\neg a \in [o]_s$ and a occurs in d for some $d \rhd f \in c'$ or occurs positively in p', or*
(4) *$\neg a \in [o']_s$ and a occurs in d for some $d \rhd f \in c$ or occurs positively in p.*

Lemma 2.10 *Let s be a state and o and o' two operators. If o and o' interfere in s, then o and o' interfere.*

PROOF. Definition of interference has the form that o and o' interfere if there is an effect (conditional or unconditional) that fulfils some property. Interference in s is the same, except that a restriction to the subclass of effects active in s is made.

As an example we consider one case. Other cases are analogous. So assume o and o' interfere in s because (case (1)) there is $a \in A$ such that $a \in [o]_s$ and a occurs negatively in the precondition of o'. Now case (1) of the definition of interference is fulfilled because there is $a \in A$ that is an active effect of o and occurs negatively in the precondition of o'. □

Lemma 2.11 *Let s be a state and S a set of operators so that $app_S(s)$ is defined and no two operators interfere in s. Then $app_S(s) = app_{o_1;\ldots;o_n}(s)$ for any total ordering $o_1, \ldots, o_n$ of S.*

PROOF. Let $o_1, \dots, o_n$ be any total ordering of S. We prove by induction on the length of a prefix of $o_1, \dots, o_n$ the following statement for all $i \in \{0, \dots, n-1\}$ by induction on i: $s \models a$ if and only if $app_{o_1;\dots;o_i}(s) \models a$ for all state variables a occurring in an antecedent of a conditional effect or a precondition of operators $o_{i+1}, \dots, o_n$.

Base case $i = 0$: Trivial.

Inductive case $i \geq 1$: By the induction hypothesis the antecedents of conditional effects of o_i have the same value in s and in $app_{o_1;\dots;o_{i-1}}(s)$, from which follows $[o_i]_s = [o_i]_{app_{o_1;\dots;o_{i-1}}(s)}$. Since o_i does not interfere in s with operators $o_{i+1}, \dots, o_n$, no state variable occurring in $[o_i]_s$ occurs in an antecedent of a conditional effect or in the precondition of $o_{i+1}, \dots, o_n$. Hence these state variables do not change. Since $[o_i]_s = [o_i]_{app_{o_1;\dots;o_{i-1}}(s)}$, this also holds when o_i is applied in $app_{o_1;\dots;o_{i-1}}(s)$. This completes the induction proof.

Since $app_S(s)$ is defined, the precondition of every $o \in S$ is true in s and $[o]_s$ is consistent. Based on the fact we have established above, the precondition of every $o \in S$ is true also in $app_{o_1;\dots;o_k}(s)$ and $[o]_{app_{o_1;\dots;o_k}(s)}$ is consistent for any $\{o_1, \dots, o_k\} \subseteq S \backslash \{o\}$. Hence any total ordering of the operators is executable. Based on the fact we have established above, $[o]_s = [o]_{app_{o_1;\dots;o_k}(s)}$ for every $\{o_1, \dots, o_k\} \subseteq S \backslash \{o\}$. Hence every operator causes the same changes no matter what the total ordering is. Since $app_S(s)$ is defined, no operator in S undoes the effects of another operator. Hence the same state $s' = app_S(s)$ is reached in every case. □

Theorem 2.12 *Let I be a state, O a set of operators, and $T = \langle S_0, \dots, S_{l-1} \rangle \in (2^O)^l$ such that there is a sequence $s_0, s_1, \dots, s_l$ of states with $s_0 = I$ and $s_{i+1} = app_{S_i}(s_i)$ for all $i \in \{0, \dots, l-1\}$. If for no $i \in \{0, \dots, l-1\}$ and $\{o, o'\} \subseteq S_i$ such that $o \neq o'$ the operators o and o' interfere in s_i, then T is a $\forall$-step plan for O and I.*

PROOF. Directly by Lemma 2.11. □

Theorem 2.13 *Let I be a state, O a set of operators, and $T = \langle S_0, \dots, S_{l-1} \rangle \in (2^O)^l$ such that there is a sequence $s_0, s_1, \dots, s_l$ of states with $s_0 = I$ and $s_{i+1} = app_{S_i}(s_i)$ for all $i \in \{0, \dots, l-1\}$. If for no $i \in \{0, \dots, l-1\}$ and $\{o, o'\} \subseteq S_i$ such that $o \neq o'$ the operators o and o' interfere, then T is a $\forall$-step plan for O and I.*

PROOF. By Lemma 2.10 and Theorem 2.12. □

The state-dependent definition of interference in some cases allows more parallelism than the state-independent definition.

Example 2.14 Consider $S = \{\langle \top, \emptyset, \{a \rhd \{\neg b\}\}\rangle, \langle \top, \emptyset, \{b \rhd \{\neg a\}\}\rangle\}$. The operators interfere according to Definition 2.6. However, the operators do not interfere in states s such that $s \models \neg a \wedge \neg b$ because no effect is active. ■

A still more relaxed notion of interference that allows changing shared state variables as long as the preconditions do not become false nor the values of antecedents of conditional effects change leads to high complexity because states other than the current one have to be considered. Even if none of the operators change the values of antecedents of conditional effects or preconditions in the current state, they may do this in states reachable by applying another operator. For example, the operator $\langle a \vee b, \{c\}, \emptyset \rangle$ is not disabled by $\langle \top, \{\neg a\}, \emptyset \rangle$

nor $\langle \top, \{\neg b\}, \emptyset \rangle$ alone, but in states reached by one of these operators the other operator disables it.

The source of the high complexity of the general definition is that on different execution orders, all of which must result in the same state, a different sequence of intermediate states is visited, and it seems unavoidable to make these intermediate states explicit when reasoning about the executions.

2.2 Process semantics

The idea of the process semantics is that we only consider those $\forall$-step plans that fulfil the following condition. There is no operator o applied at time $t+1$ with $t \geq 0$ such that the sequence of sets of operators obtained by moving o from time $t+1$ to time t would be a $\forall$-step plan that leads to the same state.

As an example consider a set S of operators that are all initially executable and no two operators interfere or have contradicting effects. If we have time points 0 and 1, we can apply each operator alternatively at 0 or at 1. The resulting state at time point 2 will be the same in all cases. So, under $\forall$-step semantics the number of equivalent plans on two time points is $2^{|S|}$. Process semantics says that no operator that is executable at 0 may be applied later than at 0. Hence under process semantics there is only one plan instead of $2^{|S|}$.

The idea of the process semantics was previously investigated in connection with Petri nets [Best and Devillers 1987]. It can be seen as a way of canonizing $\forall$-step executions into a normal form in which each operator of the $\forall$-step plan occurs as early as possible. This canonical normal form is similar to the Foata normal form in the theory of Mazurkiewicz traces [Diekert and Métivier 1997; Heljanko 2001].

Definition 2.15 (Process plans) *For a set of operators O and an initial state I* a process plan for O and *I is a $\forall$-step plan $\langle S_0, \ldots, S_{l-1} \rangle$ for O and I with the execution $s_0, \ldots, s_l$ such that there is no $i \in \{1, \ldots, l-1\}$ and $o \in S_i$ so that $\langle S_0, \ldots, S_{i-1} \cup \{o\}, S_i \backslash \{o\}, \ldots, S_{l-1} \rangle$ is a $\forall$-step plan for O and I with the execution $s'_0, \ldots, s'_l$ such that $s_j = s'_j$ for all $j \in \{0, \ldots, i-1, i+1, \ldots, l\}$.*

Note that it is possible that $o \in S_{i-1}$, and when transforming a $\forall$-step plan to a corresponding process plan, the number of operators in the plan may decrease. It is possible to define an alternative process semantics so that moving an operator earlier is possible only if the total number of operators is preserved.

The important property of process semantics is that even though the additional condition reduces the number of valid plans, whenever there is a plan with t time steps under $\forall$-step semantics, there is also a plan with at most t time steps under process semantics that leads to the same final state. From any $\forall$-step plan a plan satisfying the process condition is obtained by repeatedly moving operators violating the condition one time point earlier.

Theorem 2.16 *Let $\pi = \langle A, I, O, G \rangle$ be a problem instance and $\langle S_0, \ldots, S_{l-1} \rangle$ a $\forall$-step plan for π. Then there is a process plan $\langle S'_0, \ldots, S'_{l-1} \rangle$ for π.*

PROOF. Define a mapping ρ from plans to plans: plan $\rho(T)$ is obtained from T by moving one operator earlier according to Definition 2.15 if possible, and otherwise $\rho(T) = T$. Define the function $f(\langle S_0, \ldots, S_{l-1} \rangle) = \sum_{i=0}^{l-1}(i \cdot |S_i|)$. Note that $f(\rho(T)) < f(T)$ if $\rho(T) \neq T$. Since f can take only positive values, only finitely many moves are possible.

When $f(\rho(T)) = f(T)$, T is a process plan. Hence a process plan is obtained after finitely many moves. □

Theorem 2.17 *Testing whether a sequence of sets of operators is a process plan is polynomial-time reducible to testing whether a sequence of sets of operators is a $\forall$-step plan.*

PROOF. The definition of process plans gives a procedure for doing the test. Consider $\langle S_0, \dots, S_{l-1}\rangle$. For every operator in $S_1 \cup \dots \cup S_{l-1}$ we have to test the process condition. There are $|S_1| + \dots + |S_{l-1}|$ such tests. □

We will later concentrate on $\forall$-step plans in which no two simultaneous operators interfere, and hence it is convenient to define a narrower class of process plans that is compatible with this narrower class of $\forall$-step plans.

Definition 2.18 (i-Process plans) *For a set of operators O and an initial state I* a process plan for O and I *is a $\forall$-step plan $\langle S_0, \dots, S_{l-1}\rangle$ for O and I with the execution $s_0, \dots, s_l$ such that there is no $i \in \{1, \dots, l-1\}$ and $o \in S_i$ so that $\langle S_0, \dots, S_{i-1} \cup \{o\}, S_i \backslash \{o\}, \dots, S_{l-1}\rangle$ is a $\forall$-step plan for O and I with the execution $s'_0, \dots, s'_l$ such that $s_j = s'_j$ for all $j \in \{0, \dots, i-1, i+1, \dots, l\}$ and additionally, for no $i \in \{0, \dots, l-1\}$ and $\{o, o'\} \in S_i$ such that $o \neq o'$ the operators o and o' interfere.*

2.3 $\exists$-Step semantics

We present a general formalization of a notion of parallel plans that was first considered by Dimopoulos et al. [1997].

Definition 2.19 ($\exists$-Step plans) *For a set O of operators and an initial state I,* a $\exists$-step plan *is $T = \langle S_0, \dots, S_{l-1}\rangle \in (2^O)^l$ together with a sequence of states $s_0, \dots, s_l$ (the execution of T) for some $l \geq 0$ such that*

(1) *$s_0 = I$, and*

(2) *for every $i \in \{0, \dots, l-1\}$ there is a total ordering $o_1 < \dots < o_n$ of S_i such that $s_{i+1} = app_{o_1;\dots;o_n}(s_i)$.*

The difference to $\forall$-step semantics is that instead of requiring that each step S_i can be ordered to any total order, it is sufficient that there is one order that maps state s_i to s_{i+1}. Unlike in $\forall$-step semantics, the successor s_{i+1} of s_i is not uniquely determined solely by S_i, as the successor depends on the implicit ordering of S_i. Hence the definition has to make the execution $s_0, \dots, s_l$ explicit. There are also other important technical differences between $\exists$-step and $\forall$-step semantics, most notably the fact that the properties given in Lemma 2.2 for $\forall$-step semantics do not hold for $\exists$-step semantics.

The more relaxed definition of $\exists$-step plans sometimes allows much more parallelism than the definition of $\forall$-step plans.

Example 2.20 Consider a row of n Russian dolls, each slightly bigger than the preceding one. We can nest all the dolls by putting the first inside the second, then the second inside the third, and so on, until every doll except the biggest one is inside another doll.

For four dolls this can be formalized as follows.

$$\begin{aligned} o_1 &= \langle \text{out1} \wedge \text{out2} \wedge \text{empty2}, \{\text{1in2}, \neg\text{out1}, \neg\text{empty2}\}, \emptyset\rangle \\ o_2 &= \langle \text{out2} \wedge \text{out3} \wedge \text{empty3}, \{\text{2in3}, \neg\text{out2}, \neg\text{empty3}\}, \emptyset\rangle \\ o_3 &= \langle \text{out3} \wedge \text{out4} \wedge \text{empty4}, \{\text{3in4}, \neg\text{out3}, \neg\text{empty4}\}, \emptyset\rangle \end{aligned}$$

The shortest $\forall$-step plan that nests the dolls is $\langle\{o_1\}, \{o_2\}, \{o_3\}\rangle$. The $\exists$-step plan $\langle\{o_1, o_2, o_3\}\rangle$ nests the dolls in one step. ■

Theorem 2.21 *(i) Each $\forall$-step plan is a $\exists$-step plan, and (ii) for every $\exists$-step plan T there is a $\forall$-step plan whose execution leads to the same final state as that of T.*

PROOF. (i) Consider a $\forall$-step plan $T = \langle S_0, \dots, S_{l-1}\rangle$. Any total ordering of $S_i, i \in \{0, \dots, l-1\}$ takes state s_i to the same s_{i+1}. Hence, T is a $\exists$-step plan. (ii) For a $\exists$-step plan $T = \langle S_0, \dots, S_{l-1}\rangle$, a $\forall$-step plan whose execution leads to the same final state as that of T is $\{o_1^0\}, \dots, \{o_{n_0}^0\}, \dots, \{o_1^{l-1}\}, \dots, \{o_{n_{l-1}}^{l-1}\}$ where for every $i \in \{0, \dots, l-1\}$, the sequence $\{o_1^i\}, \dots, \{o_{n_i}^i\}$ is a total ordering of S_i given by Condition 2 of Definition 2.19. □

Next we identify restricted intractable and tractable classes of $\exists$-step plans.

Theorem 2.22 *Let O be a set of operators and I a state. Testing whether $T = \langle S_0, \dots, S_{l-1}\rangle \in (2^O)^l$ is a $\exists$-step plan for O and I with some execution $s_0, \dots, s_l$ is NP-hard, even when the set of atomic effects of operators in S_i for every $i \in \{0, \dots, l-1\}$ is consistent.*

PROOF. By reduction from SAT. Let ϕ be any propositional formula. Let A be the set of propositional variables occurring in ϕ. Let s and s' be states such that $s \not\models a$ for all $a \in A$ and $s' \models a$ for all $a \in A$. We claim that ϕ is satisfiable if and only if $\langle S\rangle$ with $S = \{\langle\top, \{a\}, \emptyset\rangle | a \in A\} \cup \{\langle\phi, \emptyset, \emptyset\rangle\}$ is a $\exists$-step plan with execution s, s'.

So assume ϕ is satisfiable and $v : A \to \{0, 1\}$ is a valuation satisfying ϕ. Then for any total order on S such that exactly the operators $S_v = \{\langle\top, \{a\}, \emptyset\rangle | a \in A, v(a) = 1\}$ precede $o_\phi = \langle\phi, \emptyset, \emptyset\rangle$ satisfies the definition of $\exists$-step plans because executing S_v produces the state/valuation v that satisfies the precondition of o_ϕ.

Assume $\langle S\rangle$ is a $\exists$-step plan. Hence there is a total ordering $o_1, \dots, o_n$ of S such that $app_{o_1;\dots;o_n}(s)$ is defined. Hence $app_{o_1;\dots;o_j}(s) \models \phi$ where $o_1, \dots, o_j$ are the operators preceding o_ϕ. Therefore ϕ is satisfiable. □

The preceding theorem (Theorem 2.22) and the following (Theorem 2.23) can be strengthened so that all operators in S_i are executable in s_i. This shows that our later restriction to sets S_i so that $app_{S_i}(s_i)$ is defined does not directly reduce complexity.

From the above proof we see that NP-hardness holds even when there are no conditional effects and the effects of the operators are not in conflict with each other. However, the proof assumes disjunctivity in preconditions because ϕ may be any formula. The question arises if the problem is easier for STRIPS operators.

Theorem 2.23 *Let O be a set of STRIPS operators and I a state. Testing whether $T = \langle S_0, \dots, S_{l-1}\rangle \in (2^O)^l$ is a $\exists$-step plan for O and I with some execution $s_0, \dots, s_l$ is NP-hard.*

PROOF. We reduce the NP-complete problem SAT to testing whether a sequence of sets of operators is a $\exists$-step plan. Let C be a set of clauses, $n = |C|$ and P the set of propositional variables occurring in C. Assign an index $i \in \{1, \dots, n\}$ to each clause. The state variables are $A = \{c_1, \dots, c_n\} \cup \{U_a | a \in P\}$. Define

$$\begin{aligned}
o_a^+ &= \langle U_a, \{\neg U_a, c_{i_1^{a+}}, \dots, c_{i_{m_{a+}}^{a+}}\}, \emptyset\rangle \text{ for all } a \in P,\\
&\quad \text{where } i_1^{a+}, \dots, i_{m_{a+}}^{a+} \text{ are the indices of clauses in which } a \text{ occurs positively}\\
o_a^- &= \langle U_a, \{\neg U_a, c_{i_1^{a-}}, \dots, c_{i_{m_{a-}}^{a-}}\}, \emptyset\rangle \text{ for all } a \in P,\\
&\quad \text{where } i_1^{a-}, \dots, i_{m_{a-}}^{a-} \text{ are the indices of clauses in which } a \text{ occurs negatively}\\
o_m &= \langle c_1 \wedge \cdots \wedge c_n, \{U_a | a \in P\}, \emptyset\rangle, \text{ and}\\
S &= \{o_a^+ | a \in A\} \cup \{o_a^- | a \in P\} \cup \{o_m\}.
\end{aligned}$$

Let s and s' be states such that $s \models \neg c_1 \wedge \cdots \wedge \neg c_n \wedge \bigwedge_{a \in P} U_a$ and $s' \models c_1 \wedge \cdots \wedge c_n \wedge \bigwedge_{a \in P} \neg U_a$. We show that $\langle S\rangle$ is a $\exists$-step plan with execution s, s' if and only if C is satisfiable. Assume that $v : P \to \{0, 1\}$ is a valuation that satisfies C. Take any total ordering $<$ of S such that for all $a \in P$, $o_a^+ < o_m$ iff $v(a) = 1$ and $o_a^- < o_m$ iff $v(a) = 0$. Applying the operators preceding o_m makes the state variables $c_1, \dots, c_n$ true (because v is a valuation that satisfies C) and the state variables $U_a, a \in P$ false. Now o_m is executable and its application makes all $U_a, a \in P$ true again. Then the remaining operators are executable, making every $U_a, a \in P$ false. Hence that total ordering satisfies the definition of $\exists$-step plans for $\langle S\rangle$ with execution s, s'.

For the other direction, assume that $\langle S\rangle$ is a $\exists$-step plan with execution s, s' which means that the operators can be applied in some order $<$ to obtain s' from s. Since for every $a \in P$ the operators o_a^+ and o_a^- have U_a as the precondition and both make U_a false and only o_m can make U_a true, it must be that $o_a^+ < o_m < o_a^-$ or $o_a^- < o_m < o_a^+$. Define $v : P \to \{0, 1\}$ by $v(a) = 1$ iff $o_a^+ < o_m$. For o_m to be executable $c_1 \wedge \cdots \wedge c_n$ must be true. Hence the operators applied before o_m correspond to a valuation v that satisfies every clause in C. Therefore $v \models C$. $\square$

Restrictions of the previous two theorems separately do not yield tractability, but together they do.

Theorem 2.24 *Let O be a set of STRIPS operators and I a state. Testing whether $T = \langle S_0, \dots, S_{l-1}\rangle \in (2^O)^l$ with no S_i containing operators with mutually conflicting effects, is a $\exists$-step plan for O and I with some execution $s_0, \dots, s_l$ is polynomial time.*

PROOF. Since no two simultaneous operators have effects that conflict each other the execution of the plan – if one exists – is unambiguously determined by the sets of effects of operators of $S_0, \dots, S_{l-1}$: $s_0 = I$ and $s_{i+1} = \mathit{app}_{\{\langle \top, e, \emptyset\rangle | \langle p, e, \emptyset\rangle\}}(s_i)$ for all $i \in \{0, \dots, l-1\}$. The question that we must answer in polynomial time is whether the operators at each time point can be ordered so that the precondition is satisfied when an operator is applied.

The test is performed by the procedure calls linearize(s_i, S_i) for all $i \in \{0, \dots, l - 1\}$. This procedure is given in Figure 1. It runs in polynomial time in the size of S because the number of iterations of the *while* loop is bounded by the cardinality of S and all the computation in one iteration is polynomial time in the size of S. We show that the procedure returns *true* if and only if an executable ordering of S exists.

```
1:  procedure linearize($s$,$S$)
2:  while $S \neq \emptyset$ do
3:      if there is $o = \langle p, e, \emptyset \rangle \in S$
4:          such that $s \models p$ and $e \cap \{\bar{l} | l \in p'\} = \emptyset$ for all $\langle p', e', \emptyset \rangle \in S \backslash \{o\}$
5:      then $S$ := $S \backslash \{o\}$
6:      else return false;
7:      $s$ := $app_o(s)$;
8:  end while
9:  return true;
```

Fig. 1. Algorithm for testing whether a set of non-conflicting STRIPS operators can be linearized

Assume linearize(s,S) returns *true*. Hence there is a sequence of states $s'_0, \ldots, s'_{|S|}$ and a sequence $o'_0, \ldots, o'_{|S|-1}$ of operators such that $s'_0 = s$ and $s'_{i+1} = app_{o'_i}(s'_i)$ for every $i \in \{0, \ldots, |S| - 1\}$. Hence $app_{o'_0;\ldots;o'_{|S|-1}}(s) = app_S(s)$ which satisfies the conditions a set S has to satisfy in the definition of $\exists$-step plans.

Assume linearize(s,S) returns *false*. We show that no execution exists. Since *false* is returned, for every $\langle p, e, \emptyset \rangle \in S' \subseteq S$ either $s' \not\models p$ (where S' and s' are the last values the variables S and s have obtained) or e falsifies the precondition of at least one of the operators in $S' \backslash \{\langle p, e, \emptyset \rangle\}$. Let $o_1, \ldots, o_n$ be any total ordering of S. We show that $app_{o_1;\ldots;o_n}(s)$ is not defined, and hence the total ordering does not satisfy Definition 2.19.

Take the operator $o_i = \langle p_i, e_i, \emptyset \rangle \in S'$ that comes earliest in the ordering $o_1, \ldots, o_n$.

If $s'_i = app_{o_1;\ldots;o_{i-1}}(s)$ is not defined (because the precondition of one of the operators is false when the operator is applied), then also $app_{o_1;\ldots;o_n}(s)$ is not defined. So assume $s'_i = app_{o_1;\ldots;o_{i-1}}(s)$ is defined.

Since linearize(s,S) returns *false*, either $s' \not\models p_i$ or o_i falsifies the precondition of at least one of $S' \backslash \{o_i\}$.

In the first case, as none of the operators in $S \backslash S'$ falsifies any literal in the precondition of any operator in S', it must be that $s \not\models p_i$. Since $s' \not\models p_i$, there is at least one conjunct (a literal) of p_i that is not made true by any operator in $S \backslash S'$. Since $\{o_1, \ldots, o_{i-1}\} \subseteq S \backslash S'$, this literal is also not true in s'_i and hence $s'_i \not\models p_i$.

In the second case, as o_i is the first operator of S_i in the ordering, one of the literals in the precondition of at least one operator in $S' \backslash \{o_i\}$ becomes false when o_i is applied. Since the operators in S are pairwise non-conflicting, there is no operator that could make this literal and precondition true again (here we use the assumption that S consists of STRIPS operators.) Hence $app_{o_1;\ldots;o_n}(s)$ is not defined, and the definition of $\exists$-step plans is not satisfied. $\square$

To obtain a tractable notion of $\exists$-step plans for operators in general we introduce, similarly to $\forall$-step semantics, a syntactic notion characterizing dependencies between operators that leads to a simple graph-theoretic test for plans.

Our quest for tractable notions of $\exists$-step plans is motivated by the need to effectively encode the planning problem in the propositional logic (Section 3.) Even though Theorem 2.24 allows $\exists$-step plans in which the preconditions of some of the operators in S_i are false in s_i, we will not consider encodings of this generality. Allowing this would seem to require making the implicit intermediate states explicit, which would directly contradict the motivation of studying parallel encodings in the first place.

Definition 2.25 (Affect) *Let A be a set of state variables and $o = \langle p, e, c\rangle$ and $o' = \langle p', e', c'\rangle$ operators over A. Then o* affects *o' if there is $a \in A$ such that*

(1) *$a \in (e \cup \bigcup\{d | f \rhd d \in c\})$ and a occurs in f for some $f \rhd d \in c'$ or occurs negatively in p', or*

(2) *$\neg a \in e$ or $\neg a \in d$ for some $f \rhd d \in c$ and a occurs in f for some $f \rhd d \in c'$ or occurs positively in p'.*

This is like Definition 2.6 but considers only one direction of interference: if o and o' interfere, then either o affects o' or o' affects o.

Lemma 2.26 *Let $o_1 < \cdots < o_n$ be an ordering of a set S of operators so that if $o < o'$ then o does not affect o'. Let s be a state so that $s \models p$ and $[o]_s$ is consistent for every $\langle p, e, c\rangle \in S$. Then the following hold.*

(1) *$app_{o_1;\ldots;o_i}(s) \models p_j$ for every $i \in \{1, \ldots, n-1\}$ and $j \in \{i+1, \ldots, n\}$ where p_j is the precondition of o_j.*

(2) *$[o_j]_s = [o_j]_{app_{o_1;\ldots;o_i}(s)}$ for every $i \in \{1, \ldots, n-1\}$ and $j \in \{i+1, \ldots, n\}$.*

(3) *For every $i \in \{1, \ldots, n\}$, if $app_{\{o_1,\ldots,o_i\}}(s)$ is defined, then $app_{o_1;\ldots;o_i}(s) = app_{\{o_1,\ldots,o_i\}}(s)$.*

PROOF. By induction on i.

Base case $i = 0$: Trivial.

Inductive case $i \geq 1$: First we note that $app_{o_1;\ldots;o_i}(s)$ is defined because by the induction hypothesis for case (1) the precondition of o_i is true in $app_{o_1;\ldots;o_{i-1}}(s)$, and by the assumptions and the induction hypothesis for case (2) $[o_i]_{app_{o_1;\ldots;o_{i-1}}(s)}$ is consistent.

Now consider any $j \in \{i+1, \ldots, n\}$.

Case (1): By the induction hypothesis $app_{o_1;\ldots;o_{i-1}}(s) \models p_j$. Since o_i does not affect o_j, o_i does not falsify p_j. Hence $app_{o_1;\ldots;o_i}(s) \models p_j$.

Case (2): By the induction hypothesis $[o_j]_s = [o_j]_{app_{o_1;\ldots;o_{i-1}}(s)}$. Since o_i does not affect o_j, o_i does not change the value of any state variable occurring in the antecedent of a conditional effect of o_j. Hence $[o_j]_s = [o_j]_{app_{o_1;\ldots;o_i}(s)}$.

Case (3): By the induction hypothesis, if $app_{\{o_1,\ldots,o_{i-1}\}}(s)$ is defined, then $app_{o_1;\ldots;o_{i-1}}(s) = app_{\{o_1,\ldots,o_{i-1}\}}(s)$. So assume also $app_{\{o_1,\ldots,o_i\}}(s)$ is defined, that is, $[o_i]_s$ does not contradict $[\{o_1, \ldots, o_{i-1}\}]_s$. By (2) $[o_i]_s = [o_i]_{app_{o_1;\ldots;o_{i-1}}(s)}$. Since the effects of o_i do not override the effects of any operator earlier in the sequence, we get $app_{o_1;\ldots;o_i}(s) = app_{\{o_1,\ldots,o_i\}}(s)$. $\square$

Theorem 2.27 *Let O be a set of operators, I a state, $T = \langle S_0, \ldots, S_{l-1}\rangle \in (2^O)^l$, and $s_0, \ldots, s_l$ a sequence of states. If*

(1) *$s_0 = I$,*

(2) *for every $i \in \{0, \ldots, l-1\}$ there is a total ordering $<$ of S_i such that if $o < o'$ then o does not affect o', and*

(3) *$s_{i+1} = app_{S_i}(s_i)$ for every $i \in \{0, \ldots, l-1\}$,*

then T is a $\exists$-step plan for O and I.

PROOF. Since by assumption $app_{S_i}(s_i)$ is defined, the preconditions of all operators in S_i are true in s_i and $[o]_{s_i}$ is consistent for every $o \in S_i$. Hence the assumptions of Lemma 2.26 are satisfied and by (3) $app_{o_1;\ldots;o_n}(s_i) = app_{S_i}(s_i)$ for some total ordering $o_1, \ldots, o_n$ of S_i. $\Box$

For STRIPS operators the subclass of $\exists$-step plans definable by using the notion of *affects* in Theorem 2.27 is not very restrictive. In comparison to arbitrary $\exists$-step plans, the only restrictions are that sets S of simultaneous operators have no contradicting effects and all operators are executable in the current state s, or in other words, that $app_S(s)$ is defined. This is stated in the following theorem.

Theorem 2.28 *Let $\pi = \langle A, I, O, G\rangle$ be a problem instance so that every operator in O is a STRIPS operator and let $T = \langle S_0, \ldots, S_{l-1}\rangle$ be a $\exists$-step plan for π with execution $s_0, \ldots, s_l$ so that $s_0 = I$ and $s_{i+1} = app_{S_i}(s_i)$ for every $i \in \{0, \ldots, l-1\}$. Then for every $i \in \{0, \ldots, l-1\}$ there is a total ordering $<$ of S_i such that if $o < o'$ then o does not affect o'.*

PROOF. For STRIPS operators an operator o *affects* o' if and only if o has an effect m and $\overline{m}$ is one of the conjuncts in the precondition of o'. The result follows from the proof of Theorem 2.24. The procedure *linearize* repeatedly selects an operator that does not affect any of the remaining operators. $\Box$

Even though the class of $\exists$-step plans based on *affects* is narrower than the class sanctioned by Definition 2.19, much more parallelism is still possible in comparison to the class of $\forall$-step plans satisfying the non-interference condition. For instance, nesting of Russian dolls in Example 2.20 belongs to this class.

Similarly to the notion of interference in a state (Definition 2.9), we could define a state-specific notion of *affects*. This would lead to a slightly more relaxed but still efficient test of whether $\exists$-step semantics is fulfilled.

It is possible to combine the $\exists$-step semantics and the process semantics, but we leave this to future work.

3. PLANNING AS SATISFIABILITY

Planning as satisfiability was introduced by Kautz and Selman [1992]. In addition to being a powerful approach to planning, it is also the basis of *bounded model checking* [Biere, Cimatti, Clarke, and Zhu 1999][1].

In this section we present encodings of the different semantics of parallel plans in the propositional logic. A basic assumption in all these encodings is that for sets S of simultaneous operators applied in state s the state $app_S(s)$ is defined, that is, all the preconditions are true in s and the set of active effects of the operators is consistent. Given this assumption, the encodings of all the semantics share a common part which is described next.

3.1 The base encoding

Planning can be performed by propositional satisfiability testing as follows. Produce formulae $\phi_0, \phi_1, \phi_2, \ldots$ such that ϕ_l is satisfiable iff there is a plan of length l. The formulae

[1]Bounded model checking was developed at CMU after Alessandro Cimatti gave there a seminar talk on the techniques used in the 1998 AIPS planning competition in which the BLACKBOX planner by Kautz and Selman participated [Cimatti 2003].

are tested for satisfiability in the order of increasing plan length, and from the satisfying assignment that is found a plan is constructed. The encodings of the different semantics for parallel plans differ only in the formulae that restrict the simultaneous application of operators. Next we describe the part of the encodings that is shared by all of the semantics.

For the problem instance $\pi = \langle A, I, O, G\rangle$ let the (Boolean) state variables be $A = \{a^1, \ldots, a^n\}$ and the operators $O = \{o^1, \ldots, o^m\}$. For every state variable $a \in A$ we have the propositional variables a_t which express the value of a at different time points $t \in \{0, \ldots, l\}$. Similarly, for every operator $o \in O$ we have o_t for expressing whether o is applied at $t \in \{0, \ldots, l-1\}$. For formulae ϕ about the values of the state variables we denote the formula with all state variables subscripted with the index to a time point t by ϕ_t.

Given a problem instance $\pi = \langle A, I, O, G\rangle$, a formula $\Phi_{\pi,l}$ is generated to answer the following question. Is there an execution of a sequence of l sets of operators from O that reaches a state satisfying G from the initial state I? The formula $\Phi_{\pi,l}$ is conjunction of I_0 (formula describing the initial state with propositional variables subscripted by time point 0), G_l, and the formulae described below, instantiated with all $t \in \{0, \ldots, l-1\}$.

First, for every $o = \langle p, e, c\rangle \in O$ there are the following formulae. The precondition p has to be true when the operator is applied.

$$o_t \rightarrow p_t \tag{1}$$

If o is applied, then its unconditional effects e are true at the next time point.

$$o_t \rightarrow e_{t+1} \tag{2}$$

Here we view sets e of literals as conjunctions of literals. For every $f \rhd d \in c$ the effects d will be true if f is true at the preceding time point.

$$(o_t \wedge f_t) \rightarrow d_{t+1} \tag{3}$$

Second, the value of a state variable does not change if no operator that changes it is applied. Hence for every state variable a we have two formulae, one expressing the conditions for the change of a from true to false,

$$(a_t \wedge \neg a_{t+1}) \rightarrow ((o^1_t \wedge (\mathit{EPC}_{\neg a}(o^1))_t) \vee \cdots \vee (o^m_t \wedge (\mathit{EPC}_{\neg a}(o^m))_t)), \tag{4}$$

and another from false to true,

$$(\neg a_t \wedge a_{t+1}) \rightarrow ((o^1_t \wedge (\mathit{EPC}_{a}(o^1))_t) \vee \cdots \vee (o^m_t \wedge (\mathit{EPC}_{a}(o^m))_t)). \tag{5}$$

These formulae can be simplified by using the obvious equivalences when $\mathit{EPC}_{\neg a}(o) = \bot$.

The formulae $\Phi_{\pi,l}$, just like the definition of $\mathit{app}_S(s)$, allow sets of operators in parallel that do not correspond to any sequential plan. For example, the operators $\langle a, \{\neg b\}, \emptyset\rangle$ and $\langle b, \{\neg a\}, \emptyset\rangle$ may be executed simultaneously resulting in a state satisfying $\neg a \wedge \neg b$, even though this state is not reachable by the two operators sequentially. Plans following the three semantics of parallel plans can always be executed sequentially. Further formulae that are discussed in the next sections are needed for capturing the three semantics.

Theorem 3.1 *Let $\pi = \langle A, I, O, G\rangle$ be a problem instance. Then there is $T = \langle S_0, \ldots, S_{l-1}\rangle \in (2^O)^l$ so that $s_0, \ldots, s_l$ are states so that $I = s_0$, $s_l \models G$, and $s_{i+1} = \mathit{app}_{S_i}(s_i)$ for all $i \in \{0, \ldots, l-1\}$ if and only if there is a valuation satisfying the formula $\Phi_{\pi,l}$.*

PROOF. For the proof from left to right, we construct a valuation v as follows. For all $i \in \{0, \ldots, l\}$ and all state variables $a \in A$ define $v(a_i) = s_i(a)$. For all $i \in \{0, \ldots, l-1\}$ and all operators $o \in O$ define $v(o_i) = 1$ iff $o \in S_i$.

We show that $v \models \Phi_{\pi,l}$. From this it directly follows that $v \models I_0 \wedge G_l$. It remains to show satisfaction of instances of the schemata (1), (2), (3), (4) and (5).

(1) Consider any $i \in \{0, \ldots, l-1\}$ and $o = \langle p, e, c\rangle \in O$. If $o \notin S_i$, then $v \not\models o_i$ and immediately $v \models o_i \rightarrow p_i$ (Formula 1). So assume $o \in S_i$. By assumption s_i is a state such that $\mathit{app}_{S_i}(s_i)$ is defined. Hence the precondition of o is true in s_i. Hence $v \models o_i \rightarrow p_i$ (Formula 1).

(2) Consider any $i \in \{0, \ldots, l-1\}$ and $o = \langle p, e, c\rangle \in O$. If $o \notin S_i$, then $v \not\models o_i$ and immediately $v \models o_i \rightarrow e_{i+1}$ (Formula 2). So assume $o \in S_i$. As $o \in S_i$, the unconditional effects e of o are true in $s_{i+1} = \mathit{app}_{S_i}(s_i)$. Hence $v \models o_i \rightarrow e_{i+1}$ (Formula 2).

(3) Consider any $i \in \{0, \ldots, l-1\}$ and $o = \langle p, e, c\rangle \in O$ and $f \rhd d \in c$. If $o \notin S_i$, then $v \not\models o_i$ and immediately $v \models (o_i \wedge f_i) \rightarrow e_{i+1}$ (Formula 2). So assume $o \in S_i$. Now $v \models (o_i \wedge f_i) \rightarrow d_{i+1}$ (Formula 3) because if $s_i \models f$ then the literals d are active effects and are true in s_{i+1} and consequently $v \models d_{i+1}$.

(4) Consider any $i \in \{0, \ldots, l-1\}$ and $a \in A$. According to the definition of $s_{i+1} = \mathit{app}_{S_i}(s_i)$, a can be true in s_i and false in s_{i+1} only if $\neg a \in [o]_{s_i}$ for some $o \in S_i$. By Lemma 1.2 $\neg a \in [o]_{s_i}$ if and only if $s_i \models \mathit{EPC}_{\neg a}(o)$, where $o = \langle p, e, c\rangle$. So if the antecedent of $(a_i \wedge \neg a_{i+1}) \rightarrow ((o_i^1 \wedge (\mathit{EPC}_{\neg a}(o^1))_i) \vee \cdots \vee (o_i^m \wedge (\mathit{EPC}_{\neg a}(o^m))_i))$ is true, then one of the disjuncts of the consequent is true, where $O = \{o^1, \ldots, o^m\}$. This yields the truth of instances of Formula 4.
Proof for Formula 5 is analogous.

For the proof from right to left, assume v is a valuation satisfying the formula $\Phi_{\pi,l}$. We construct a plan $\langle S_0, \ldots, S_{l-1}\rangle$ and a corresponding execution $s_0, \ldots, s_l$.

Define for all $i \in \{0, \ldots, l\}$ the state s_i as the valuation of A such that $s_i(a) = v(a_i)$ for every $a \in A$. Define $S_j = \{o \in O | v(o_j) = 1\}$ for all $j \in \{0, \ldots, l-1\}$.

Obviously $I = s_0$ and $s_l \models G$. We show that $s_{i+1} = \mathit{app}_{S_i}(s_i)$ for all $i \in \{0, \ldots, l-1\}$.

The precondition p of every operator $o \in S_i$ is true in s_i because $v \models o_i$ and $v \models o_i \rightarrow p_i \in \Phi_{\pi,l}$ (Formula 1).

$s_{i+1} \models [o]_{s_i}$ for every $o \in S_i$ because $v \models o_i$ and $v \models o_i \rightarrow e_{i+1} \in \Phi_{\pi,l}$ for the unconditional effects e of o (Formula 2) and $v \models (o_i \wedge f_i) \rightarrow d_{i+1}$ for conditional effects $f \rhd d$ of o. This also means that $[S_i]_{s_i}$ is consistent and $\mathit{app}_{S_i}(s_i)$ is defined.

For state variables a not occurring in $[S_i]_{s_i}$ we have to show that $s_i(a) = s_{i+1}(a)$. Since a does not occur in $[S_i]_{s_i}$, for every $o \in \{o^1, \ldots, o^m\} = O$ either $o \notin S_i$ or both $a \notin [o]_{s_i}$ and $\neg a \notin [o]_{s_i}$. Hence either $v \not\models o_i$ or (by Lemma 1.2) $v \models \neg(\mathit{EPC}_a(o))_i \wedge \neg(\mathit{EPC}_{\neg a}(o))_i$. This together with the assumptions that $v \models (a_i \wedge \neg a_{i+1}) \rightarrow ((o_i^1 \wedge (\mathit{EPC}_{\neg a}(o^1))_i) \vee \cdots \vee (o_i^m \wedge (\mathit{EPC}_{\neg a}(o^m))_i))$ (Formula 4) and $v \models (\neg a_i \wedge a_{i+1}) \rightarrow ((o_i^1 \wedge (\mathit{EPC}_a(o^1))_i) \vee \cdots \vee (o_i^m \wedge (\mathit{EPC}_a(o^m))_i))$ (Formula 5) implies $v \models (a_i \rightarrow a_{i+1}) \wedge (\neg a_i \rightarrow \neg a_{i+1})$. Therefore every $a \in A$ not occurring in $[S_i]_{s_i}$ remains unchanged. Hence $s_{i+1} = \mathit{app}_{S_i}(s_i)$. $\square$

Proposition 3.2 *The size of the formula $\Phi_{\pi,l}$ is linear in l and in the size of π.*

Theorem 3.1 says that a sequence of operators fulfilling certain conditions exists if and only if a given formula is satisfiable. The theorems connecting certain formulae to certain notions of plans (Theorems 3.3, 3.6, 3.11, 3.12, 3.13) provide an implication only in one direction: whenever the formula for a given value of parameter l is satisfiable, a plan of l time points exists. The other direction is missing because the formulae in general only approximate the respective semantics and there is no guarantee that the formula for a given l is satisfiable when a plan with l time points exists. However, the formula with some higher value of l is satisfiable. This follows from the fact that whenever a $\forall$-step or $\exists$-step plan $\langle S_0, \ldots, S_{l-1}\rangle$ with $n = |S_0| + \cdots + |S_{l-1}|$ occurrences of operators exists, there is a plan consisting of n singleton sets, and the corresponding formulae $\Phi_{\pi,n} \wedge \Phi^x_{O,n}$ are satisfiable. The formulae $\Phi^x_{O,n}$ encode the parallel semantics x for formulae O.

An exact match between the $\forall$-step semantics and its encodings holds for problem instances with STRIPS operators only (Theorem 3.4.)

The implications of the approximative nature of the $\forall$-step semantics encodings for process semantics are more serious. For STRIPS operators the encodings for process semantics are exact: the formula for n time points is satisfiable if and only if a process plan of length n exists. However, in the general case the inexactness of the underlying $\forall$-step encoding leads to a mismatch between process semantics and the formulae. The problem is that the movement of an operator to an earlier time point may be prevented by the too strict $\forall$-step semantics encoding even when it is allowed by Definition 2.1. Hence the process semantics has to be understood in relation to particular classes of $\forall$-step plans: an operator has to be moved earlier only if there is a corresponding $\forall$-step plan *belonging to the subclass in question*, for example, the subclass of $\forall$-step plans in which no two parallel operators interfere. This is the reason why we introduced the notion of i-process plans in Definition 2.18.

In planning as satisfiability it is often useful to use constraints that do not affect the set of satisfying valuations but help pruning the set of incomplete solutions encountered during satisfiability testing and therefore speed up plan search. The most important type of such constraints for many planning problems is *invariants* which are formulae that are true in all states reachable from the initial state. Typically, one uses only a restricted class of invariants that are efficient (polynomial time) to identify. There are efficient algorithms for finding many invariants that are 2-literal clauses [Blum and Furst 1997; Rintanen 1998]. Theorem 3.1 does not hold if invariants are included because they contain information about the set of states that are not reachable by any sequential plan. For example, the formula $a \vee b$ is an invariant that would rule out states satisfying $\neg a \wedge \neg b$ that are reachable from any state satisfying $a \wedge b$ by simultaneous application of $\langle a, \{\neg b\}, \emptyset\rangle$ and $\langle b, \{\neg a\}, \emptyset\rangle$ but not sequentially reachable by these operators. However, the additional constraints in the following sections which restrict the parallel application of operators guarantee that only sequentially reachable states are considered. Therefore in the presence of the additional constraints for the different semantics invariants do not affect the set of satisfying valuations.

3.2 $\forall$-Step semantics

We have showed in Section 2.1 that the classes of $\forall$-step plans definable in terms of the notions of interference and interference in a state are tractable, in contrast to the general definition that is co-NP-hard.

In this section we present two encodings of the subclass of plans following $\forall$-step semantics in which no two parallel operators interfere. The first encoding is similar to the one used by Kautz and Selman in the BLACKBOX planner [Kautz and Selman 1999] and has a size that is quadratic in the number of the operators. The size of the second encoding is linear in the size of the operators. Encodings for the more relaxed notion of interference in a state can be given, including an encoding with a linear size, but we do not discuss them in detail in this work.

3.2.1 *A quadratic encoding.* The simplest encoding of the interference condition in Definition 2.6 is by formulae

$$\neg o_t \vee \neg o'_t \tag{6}$$

for every pair of interfering operators o and o'. Note that according to our definition, operators that could never be applied simultaneously (because of conflicting preconditions or effects) may interfere. The formulae (6) for these kinds of pairs of operators are of course superfluous. Define $\Phi^{\forall step,1}_{O,l}$ as the conjunction of the formulae (6) for all time points $t \in \{0, \ldots, l-1\}$ and for all pairs of interfering operators $\{o, o'\} \subseteq O$ that could be applied simultaneously. There are $\mathcal{O}(ln^2)$ such formulae for n operators.

Theorem 3.3 *Let $\pi = \langle A, I, O, G\rangle$ be a problem instance. There is a $\forall$-step plan of length l for π if $\Phi_{\pi,l} \wedge \Phi^{\forall step,1}_{O,l}$ is satisfiable.*

PROOF. Directly by Theorems 2.13 and 3.1. □

A similar quadratic-size encoding can also be given for state-dependent interference. The state-dependence is easy to encode by a formula that has a size proportional to the two operators: the simultaneous execution is allowed if none of the operators has an active effect that changes a state variable in the precondition or antecedent of a conditional effect of the other. Note that for STRIPS operators the state-dependent and state-independent notions of interference coincide, and even further, the above encoding of the $\forall$-step semantics is perfectly accurate.

Theorem 3.4 *Let $\pi = \langle A, I, O, G\rangle$ be a problem instance where O is a set of STRIPS operators. There is a $\forall$-step plan of length l for π if and only if $\Phi_{\pi,l} \wedge \Phi^{\forall step,1}_{O,l}$ is satisfiable.*

PROOF. The *if* direction is by Theorem 3.3. It remains to show the *only if* direction. So assume there is a $\forall$-step plan $T = \langle S_0, \ldots, S_{l-1}\rangle$. By Theorem 3.1 there is a valuation v such that $v \models \Phi_{\pi,l}$. We show that also $v \models \Phi^{\forall step,1}_{O,l}$, that is, any conjunct $\neg o_i \vee \neg o'_i$ of $\Phi^{\forall step,1}_{O,l}$ for $i \in \{0, \ldots, l-1\}$ and $\{o, o'\} \subseteq O$ is satisfied by v.

Since $\neg o_i \vee \neg o'_i$ is in $\Phi^{\forall step,1}_{O,l}$, o and o' interfere. By Definition 2.6 this means for operators without conditional effects that there is a literal m such that m is an effect of o and $\overline{m}$ is a conjunct of the precondition of o', or the other way round. Hence by Theorem 2.3 $\{o, o'\} \not\subseteq S_i$. By the construction of v in the proof of Theorem 3.1 $v \models \neg o_i \vee \neg o'_i$. Hence every conjunct of $\Phi^{\forall step,1}_{O,l}$ is satisfied by v. □

3.2.2 *A linear encoding.* As the size of $\Phi_{\pi,l}$ is linear in l and the size of π, the quadratic encoding of the interference constraints may dominate the size of $\Phi_{\pi,l} \wedge \Phi^{\forall step,1}_{O,l}$. We give a linear-size encoding for the interference constraints.

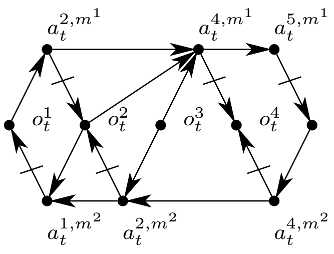


Fig. 2. A linear-size encoding of interference constraints

The idea of the encoding is to order all operators that may make a state variable $p \in A$ false (respectively true) or that have a positive (respectively negative) occurrence of p in the precondition or any occurrence in an antecedent of a conditional effect arbitrarily as $o^1, \ldots, o^n$. Whenever an operator o that falsifies p is applied, a sequence of implications prevents the application of every operator o' preceding or following o whenever o' has positive occurrences of p in the precondition or any occurrences in the antecedents of conditional effects. One chain of implications, through a set of auxiliary propositional variables, goes to the right in the ordering and another chain to the left.

We define a formula for every literal $m \in A \cup \{\neg p | p \in A\}$ for preventing the simultaneous application of operators that falsify m and operators that require m to remain true. Let $o^1, \ldots, o^n$ be any fixed ordering of the operators. Let E_m be the set of operators that may falsify m, and let R_m be the set of operators that may require m to remain true.

The formula is the conjunction of *chain*$(o^1, \ldots, o^n; E_m; R_m; m^1)$ and *chain*$(o^n, \ldots, o^1; E_m; R_m; m^2)$ for all literals m where

$$\begin{aligned}\textit{chain}(o^1, \ldots, o^n; E; R; m) \;=\; & \bigwedge \{o_t^i \rightarrow a_t^{j,m} | i < j, o^i \in E, o^j \in R, \{o^{i+1}, \ldots, o^{j-1}\} \cap R = \emptyset\} \\ & \cup \{a_t^{i,m} \rightarrow a_t^{j,m} | i < j, \{o^i, o^j\} \subseteq R, \{o^{i+1}, \ldots, o^{j-1}\} \cap R = \emptyset\} \\ & \cup \{a_t^{i,m} \rightarrow \neg o_t^i | o^i \in R\}.\end{aligned}$$

The parameter m is needed to make the names of the auxiliary variables unique. The m^1 and m^2 are two names distinguishing the auxiliary variables for the two sets of formulae for literal m.

Example 3.5 Consider the following operators.

$$\begin{aligned} o^1 &= \langle x, \{\neg x, y\}, \emptyset \rangle \\ o^2 &= \langle x, \{\neg x, z\}, \emptyset \rangle \\ o^3 &= \langle z, \{\neg x\}, \emptyset \rangle \\ o^4 &= \langle x, \{z\}, \emptyset \rangle \\ o^5 &= \langle x, \{\neg x\}, \emptyset \rangle \end{aligned}$$

The formulae that encode the constraints on the simultaneous application for these operators and the state variable x are depicted in Figure 2. ■

The number of 2-literal clauses in *chain*$(o^1, \ldots, o^n; E_m; R_m; m^i)$ is at most three times the number of operators in which m occurs and hence the number of 2-literal clauses in

$$\textit{chain}(o^1, \ldots, o^n; E_m; R_m; m^1) \wedge \textit{chain}(o^n, \ldots, o^1; E_m; R_m; m^2)$$

is at most six times the number of operators. Since we have these formulae for every literal m, the number of 2-literal clauses is linearly bounded by the size of the set of operators. Let $\Phi_{O,l}^{\forall step,2}$ be the conjunction of the above formulae for all literals m and time points $t \in \{0, \ldots, l-1\}$.

Theorem 3.6 *Let $\pi = \langle A, I, O, G\rangle$ be a problem instance. $\Phi_{\pi,l} \wedge \Phi_{O,l}^{\forall step,1}$ is satisfiable if and only if $\Phi_{\pi,l} \wedge \Phi_{O,l}^{\forall step,2}$ is satisfiable. Hence there is a $\forall$-step plan for π of length l if $\Phi_{\pi,l} \wedge \Phi_{O,l}^{\forall step,2}$ is satisfiable.*

PROOF. Let v be a valuation such that $v \models \Phi_{O,l}^{\forall step,1}$. We construct a valuation v' that satisfies $\Phi_{O,l}^{\forall step,2}$. For all variables occurring in $\Phi_{O,l}^{\forall step,1}$ we have $v'(x) = v(x)$. Additionally, v' assigns values to the auxiliary variables a_t^{i,m^1} and a_t^{i,m^2} occurring only in $\Phi_{O,l}^{\forall step,2}$.

Let $v'(a_t^{j,m^1}) = 1$ iff there is $o^i \in E_m$ such that $i < j$ and $v(o_t^i) = 1$. Let $v'(a_t^{j,m^2}) = 1$ iff there is $o^i \in E_m$ such that $i > j$ and $v(o_t^i) = 1$.

We consider only the components of the first conjunct of *chain*$(o^1, \ldots, o^n; E_m; R_m; m^1) \wedge$ *chain*$(o^n, \ldots, o^1; E_m; R_m; m^2)$. The second conjunct is analogous.

Consider $o_t^i \rightarrow a_t^{j,m^1}$ such that $i < j, o^i \in E_m, o^j \in R_m, \{o^{i+1}, \ldots, o^{j-1}\} \cap R_m = \emptyset$. If $v'(o_t^i) = 1$, then by the definition of v' also $v'(a_t^{j,m^1}) = 1$ because $i < j$ and $v'(o_t^i) = 1$.

Consider $a_t^{i,m^1} \rightarrow a_t^{j,m^1}$ such that $i < j, \{o^i, o^j\} \subseteq R_m, \{o^{i+1}, \ldots, o^{j-1}\} \cap R_m = \emptyset$. If $v(a_t^{i,m^1}) = 1$, then there is $o^{i'} \in E_m$ such that $i' < i$ and $v'(o_t^{i'}) = 1$. Therefore by the definition of v' we have $v'(a_t^{j,m^1}) = 1$.

Consider $a_t^{i,m^1} \rightarrow \neg o_t^i$ such that $o^i \in R_m$. If $v(a_t^{i,m^1}) = 1$, then there is $o^{i'} \in E_m$ such that $i' < i$ and $v'(o_t^{i'}) = 1$. Since $v' \models \neg o_t^{i'} \vee \neg o_t^i$, it must be that $v' \models \neg o_t^i$.

Hence all conjuncts of *chain*$(o^1, \ldots, o^n; E_m; R_m; m^1)$ are true in v'.

For the other direction, let v be a valuation such that $v \models \Phi_{O,l}^{\forall step,2}$. We show that $v \models \Phi_{O,l}^{\forall step,1}$. Take any conjunct $\neg o_t \vee \neg o'_t$ of $\Phi_{O,l}^{\forall step,1}$. If $v \not\models o_t$, then the truth immediately follows. Assume $v \models o_t$. Since $o = \langle p, e, c\rangle$ and $o' = \langle p', e', c'\rangle$ interfere, there is a state variable $a \in A$ that occurs as a negative effect of o and either in d for some $f \rhd d \in c'$ or positively in p' (or, the roles of o and o' are the other way around, or the polarity of the occurrences of a is complementary: the proofs of these cases are analogous.) Now $o \in E_a$ and $o' \in R_a$. We assume that the index o is lower than that of o'. The case with a higher index is analogous: instead of *chain*$(o^1, \ldots, o^n; E_a; R_a; a^1)$ we consider *chain*$(o^n, \ldots, o^1; E_a; R_a; a^2)$.

We show that because $v \models$ *chain*$(o^1, \ldots, o^n; E_a; R_a; a^1)_t$, also $v \models \neg o'_t$.

The formula *chain*$(o^1, \ldots, o^n; E_a; R_a; a^1)_t$ contains a sequence of implications $o'_t \rightarrow a_t^{j_1,a} \rightarrow a_t^{j_2,a} \rightarrow \cdots \rightarrow a_t^{j_k,a} \rightarrow \neg o_t^{j_k}$ where $o^{j_k} = o'$. Since these implications are true in v, $v \not\models o'_t$. Therefore $v \models \neg o_t \vee \neg o'_t$. Since this holds for all conjuncts of $\Phi_{O,l}^{\forall step,1}$, we have $v \models \Phi_{O,l}^{\forall step,1}$. Since $v \models \Phi_{\pi,l} \wedge \Phi_{O,l}^{\forall step,1}$ by Theorem 3.3 there is a $\forall$-step plan of length l for π. $\Box$

The number of auxiliary propositional variables is linearly proportional to the number of operators and state variables. Hence this linear-size encoding of the interference constraints may lead to formulae with a much higher number of propositional variables than with the quadratic size encoding of the constraints. The higher number of propositional variables may negatively affect the runtimes of satisfiability algorithms.

A compromise between the size of the constraints and the number of propositional variables is possible. There is an encoding of the constraints with only a logarithmic number of new propositional variables and with only $\mathcal{O}(n \log n)$ clauses which improves the quadratic encoding with respect to the number of clauses and the linear encoding with respect to the number of propositional variables. We describe the idea of the encoding without formalizing it and proving it correct.

The idea of the encoding is similar to that of *chain*$(o^1, \dots, o^n; E_m; R_m;)$ in that an arbitrary ordering is imposed on the operators and the application of an operator prevents the application of operators later in the ordering. For each literal m we encode a binary number between 0 and $|R_m| - 1$ in a logarithmic number of state variables. Then there is a formula for each operator o in E_m stating that the binary number for m has a value that is at least as high as the index of the first operator in R_m that follows o. For each operator o' in R_m there is similarly a formula that says that o' is not applied if the value of the binary number is lower than the index of o'. Hence no operator in R_m following an applied operator in E_m is applied.

The linear-size encoding and the above $n \log n$-size encoding can both be made state-dependent by observing the application of o with respect to the constraints related to literal m only if m is an active effect of o.

3.3 Process semantics

The encoding of process semantics extends the encoding of $\forall$-step semantics. We take all formulae for the latter (for example $\Phi_{\pi,l} \wedge \Phi_{O,l}^{\forall step,2}$) and have further formulae specific to process semantics.

The encoding of the underlying $\forall$-step semantics encoding and the additional constraints for process semantics are tightly coupled: when the constraints force the movement of an operator to the preceding time point, the $\forall$-step semantics constraints for the preceding time points must be compatible with the move. In this section we discuss the encoding of the process constraints for the subclass of $\forall$-step plans based on interference (Definition 2.6 and Section 3.2.) Constraints compatible with broader classes of $\forall$-step plans (for example based on Definition 2.9) are more complicated.

The formulae for process semantics prevent the application of an operator o at time $t+1$ if moving o to time t also resulted in a valid $\forall$-step plan according to Definition 2.1 and the state at time $t + 2$ stayed the same.

An operator o may be applied at time $t+1$ only if at least one of the following conditions hold.

—The precondition of o became true at $t + 1$ (and is false at t.)

—The operator o interferes with an operator at time point t (Definition 2.6.)
  This includes the following pairwise tests.
  —Could one operator falsify the precondition of the other?
  —Could one operator change the set of active effects of the other. In other words, could it change the value of the antecedent of a conditional effect of the other?
  Note that if none of the operators at t interfere with the operator at $t+1$ then the operator would have the same effects at t as it has at $t + 1$.

—The active effects of o are in conflict with the active effects of an operator at t.

We give a linear-size encoding of these conditions. Let the set of state variables be $A = \{a^1, \dots, a^n\}$. We introduce the following auxiliary propositional variables.

—The variables $a_t^{i,1}$ denote that an operator at time $t+1$ makes (may make) a^i *true*, and hence a justification for not moving the operator earlier is that
  —there is an operator at t with a *negative* occurrence of a^i in its precondition, or
  —there is an operator at t with an occurrence of a^i in the lhs of a conditional effect,

—The variables $a_t^{i,\neg 1}$ denote that an operator at time $t+1$ makes (may make) a^i *false*, and hence a justification for not moving that operator earlier is that
  —there is an operator at t with a *positive* occurrence of a^i in its precondition, or
  —there is an operator at t with an occurrence of a^i in the lhs of a conditional effect.

—The variables $a_t^{i,2}$ denote that an operator at time $t+1$ has an occurrence of a^i in the antecedent of a conditional effect, and hence a justification for not moving that operator earlier is that there is an operator at t that changes the value of a^i.

—The variables $a_t^{i,3}$ denote that an operator at time $t+1$ has a *positive* occurrence of a^i in the precondition, and hence a justification for not moving that operator earlier is that there is an operator at t that makes (may make) a^i *false*.

—The variables $a_t^{i,\neg 3}$ denote that an operator at time $t+1$ has a *negative* occurrence of a^i in the precondition, and hence a justification for not moving that operator earlier is that there is an operator at t that makes (may make) a^i *true*.

—The variables $a_t^{i,4}$ denote that an operator at time $t+1$ (actually) makes a^i *true*, and hence a justification for not moving that operator earlier is that there is an operator at t that (actually) makes a^i *false*.

—The variables $a_t^{i,\neg 4}$ denote that an operator at time $t+1$ (actually) makes a^i *false*, and hence a justification for not moving that operator earlier is that there is an operator at t that (actually) makes a^i *true*.

Note that the definition of interference in Definition 2.6 is about occurrences of a state variable in the effects of one operator and in the precondition or in the antecedents of conditional effects of another operator. This is the reason why in the above description we have stated that an operator *may make* a state variable true or false. Below we make this more explicit.

We need the following formulae for each state variable a^i and all $t \in \{0, \dots, l-1\}$.

$$a_{t+1}^{i,1} \rightarrow (o_t^1 \vee \cdots \vee o_t^n) \tag{7}$$

where $o^1, \dots, o^n$ are all the operators o that have an occurrence of a^i in the lhs of a conditional effect, or a *negative* occurrence of a^i in the precondition.

$$a_{t+1}^{i,\neg 1} \rightarrow (o_t^1 \vee \cdots \vee o_t^n) \tag{8}$$

where $o^1, \dots, o^n$ are all the operators o that have a *positive* occurrence of a^i in the precondition, or an occurrence of a^i in the lhs of a conditional effect.

$$a_{t+1}^{i,2} \rightarrow (o_t^1 \vee \cdots \vee o_t^n) \tag{9}$$

where $o^1, \dots, o^n$ are all the operators in which a^i occurs in an effect.

$$a_{t+1}^{i,3} \rightarrow (o_t^1 \vee \cdots \vee o_t^n) \tag{10}$$

where $o^1, \dots, o^n$ are all the operators o that have the effect $\neg a^i$ (possibly conditional).

$$a^{i,\neg 3}_{t+1} \rightarrow (o^1_t \vee \cdots \vee o^n_t) \tag{11}$$

where $o^1, \dots, o^n$ are all the operators o that have the effect a^i (possibly conditional).

Additionally, for each operator $o \in O$ we need a formula that lists all the possible justifications for not moving the operator one step earlier. These formulae are

$$o_t \rightarrow (\neg p_{t-1} \vee \phi) \tag{12}$$

where p is the precondition of o and ϕ is the disjunction of the propositional variables

—$a^{i,1}_t$ such that a^i is an effect (possibly conditional) of o,
—$a^{i,\neg 1}_t$ such that $\neg a^i$ is an effect (possibly conditional) of o,
—$a^{i,2}_t$ such that a^i occurs in the antecedent of a conditional effect of o,
—$a^{i,3}_t$ such that a^i occurs positively in the precondition of o, and
—$a^{i,\neg 3}_t$ such that a^i occurs negatively in the precondition of o.

For the variables $a^{i,4}_t$ and $a^{i,\neg 4}_t$ we replace each positive occurrence of a^i_t in the consequent of the implication of Formula 3 by $(a^i_t \wedge a^{i,4}_t \wedge a^{i,\neg 4}_{t-1})$ and each occurrence of $\neg a^i_t$ by $(\neg a^i_t \wedge a^{i,\neg 4}_t \wedge a^{i,\neg 4}_{t-1})$ for all $t \in \{1, \dots, l-1\}$. This is to indicate that a^i or $\neg a^i$ is an active effect of the operator at time t.

The variables $a^{i,2}_t$, $a^{i,3}_t$ and $a^{i,\neg 3}_t$ and the associated formulae are not needed if all operators are STRIPS operators. For STRIPS operators the use of variables $a^{i,4}_t$ and $a^{i,\neg 4}_t$ could be replaced by the use $a^{i,1}_t$ and $a^{i,\neg 1}_t$.

Let the formula $\Phi^{process}_{O,l}$ be a conjunction of all the above formulae. The size of $\Phi^{process}_{O,l}$ is linear in the size of the set O of operators because there are at most $2l$ variable occurrences for every state variable occurrence in every operator.

Theorem 3.7 *Let $\pi = \langle A, I, O, G \rangle$ be a problem instance. There is i-process plan T of length l for π if $\Phi_{\pi,l} \wedge \Phi^{\forall step,2}_{O,l} \wedge \Phi^{process}_{O,l}$ is satisfiable.*

PROOF. Assume v is a valuation such that $v \models \Phi_{\pi,l} \wedge \Phi^{\forall step,2}_{O,l} \wedge \Phi^{process}_{O,l}$. Define for all $i \in \{0, \dots, l\}$ the state s_i as the valuation of A such that $s_i(a) = v(a_i)$ for every $a \in A$. Define $S_j = \{o \in O | v(o_j) = 1\}$ for all $j \in \{0, \dots, l-1\}$. By Theorem 3.6 $T = \langle S_0, \dots, S_{l-1} \rangle$ is a $\forall$-step plan.

Assume that T is not an i-process plan because for some $i \in \{1, \dots, l-i\}$ and $o^x \in S_i$, $T' = \langle S_0, \dots, S_{i-1} \cup \{o^x\}, S_i \backslash \{o^x\}, \dots, S_{l-1} \rangle$ is a $\forall$-step plan in which no two simultaneous operators interfere. We show that this leads to a contradiction with the assumption that $v \models \Phi^{process}_{O,l}$.

Consider $o^x_i \rightarrow (\neg p^x_{i-1} \vee j^1 \vee \cdots \vee j^n)$. Assume that v satisfies this formula. Since $v \models o^x_i$ (as $o^x \in S_i$), at least one of the disjuncts in the right side is true in v. It cannot be that $v \models \neg p^x_{i-1}$ where p^x is the precondition of o^x because otherwise o^x would not be executable at time $i-1$ in T'.

So some other disjunct of $j^1 \vee \cdots \vee j^n$ must be satisfied by v. This leads to a long and tedious case analysis. We only give as an example the proof for the disjunct $a^{q,1}_i$ for a state variable a^q that is a positive effect of o^x. If $v \models a^{q,1}_i$, then because $v \models a^{q,1}_i \rightarrow (o^1_{i-1} \vee \cdots \vee o^n_{i-1})$ where $o^1, \dots, o^n$ are all the operators that have an occurrence of a^q in the lhs of a conditional effect or a negative occurrence in the precondition. Hence there is an operator $o^y \in S_{i-1}$ that has an occurrence of a^q in the lhs of a conditional effect or a negative occurrence in the precondition. Hence o^x and o^y interfere, and both are at $i-1$ in T', which contradicts our assumptions.

Therefore it must be the case that T is an i-process plan. $\square$

3.4 $\exists$-Step semantics

We give three encodings of the constraints that guarantee that the plans follow the $\exists$-step semantics. The first two (Sections 3.4.2 and 3.4.3) exactly encode the acyclicity test, allowing maximum parallelism with respect to a given disabling graph (as defined in Section 3.4.1). However, the first of these encodings has a cubic size and the second involves guessing a topological ordering for the set of operators, and therefore these encodings would not appear to be practical. The third encoding (Section 3.4.4) is based on assigning a fixed ordering on the operators and allowing the simultaneous application of a subset of the operators only if none of the operators affects the operators later in the ordering. The size of this encoding is linear in the size of the set of operators, but it sometimes allows less parallelism than the first two encodings. However, in our experiments this encoding has turned out to be very efficient.

To improve the efficiency of the encodings we consider a method for utilizing the structural properties of planning problems in the form of *disabling graphs* in Section 3.4.1. The idea is to identify operators for which the existence of a total ordering required by the $\exists$-step semantics can be guaranteed, no matter in which state the set of operators is simultaneously applied. The set of operators is partitioned to subsets of operators potentially involved in a cycle that cannot be totally ordered for execution. Constraints guaranteeing the ordering property need to be given only for such subsets. The decomposition method in some cases splits the set of all operators to singleton subsets. If all sets are singleton, the ordering property is guaranteed for any subset of operators applied simultaneously, and there is no need to introduce further constraints on operator application. The technique improves all the three encodings of the $\exists$-step semantics on many types of structured problems.

3.4.1 *Disabling graphs.* The motivation for using disabling graphs is the following. Define a *circularly disabled set* as a set of operators that is executable in some state and on all total orderings of the operators at least one operator affects an operator later in the ordering. Now any set-inclusion minimal circularly disabled set is a subset of a strong component (or strongly connected component, abbreviated as SCC) of the disabling graph.

Definition 3.8 *Let $\pi = \langle A, I, O, G\rangle$ be a problem instance. A graph $\langle O, E\rangle$ is a* disabling graph *for π when $E \subseteq O \times O$ is the set of directed edges so that $\langle o, o'\rangle \in E$ if*

(1) *there is a state s such that s is reachable from I by operators in O and app$_{\{o,o'\}}(s)$ is defined, and*
(2) *o affects o'.*

For a given set of operators there are typically several disabling graphs because the graph obtained by adding an edge to a disabling graph is also a disabling graph. Also the complete graph $\langle O, O \times O\rangle$ is a disabling graph. For every set of operators there is a unique minimal disabling graph, but computing minimal disabling graphs is NP-hard because of the consistency tests and PSPACE-hard because of the reachability tests of s in Condition 1. Computing non-minimal disabling graphs is easier because the consistency and reachability tests may be approximated.

We may allow the simultaneous application of a set of operators from the same SCC if the subgraph of the disabling graph induced by those operators does not contain a cycle.[2]

Lemma 3.9 *Let O be a set of operators and $G = \langle O, E\rangle$ a disabling graph for O. Let $C_1, \ldots, C_m$ be the strong components of G. Let s be a state. Let O' be a set of operators so that $app_{O'}(s)$ is defined. If for every $i \in \{1, \ldots, m\}$ the subgraph $\langle C_i \cap O', E \cap ((C_i \cap O') \times (C_i \cap O'))\rangle$ of G induced by $C_i \cap O'$ is acyclic, then there is a total ordering $o_1, \ldots, o_n$ of O' such that $app_{o_1;\ldots;o_n}(s) = app_{O'}(s)$.*

PROOF. Let the indices of $C_1, \ldots, C_m$ be such that for all $i \in \{1, \ldots, m-1\}$ and $j \in \{i+1, \ldots, m\}$ there are no edges from an operator in C_i to an operator in C_j. Such a numbering exists because the sets C_i are strong components of G (the strong components always form a tree.) Since the subgraph induced by $C_i \cap O'$ is acyclic for every $i \in \{1, \ldots, m\}$, we can impose an ordering $o_1 <_i \ldots <_i o_{n_i}$ on $C_i \cap O'$ so that if $o <_i o'$ then there is no edge from o to o', that is, o does not affect o'.

Now we can construct a total order $o_1 < \cdots < o_n$ on O' as follows. For all $\{o, o'\} \in O'$, $o < o'$ if $\{o, o'\} \subseteq C_i$ for some $i \in \{1, \ldots, m\}$ and $o <_i o'$, or $o \in C_i$ and $o' \in C_j$ and $i < j$. Now for all $\{o, o'\} \subseteq O'$, if $o < o'$ then o does not affect o'. Hence $app_{o_1;\ldots;o_n}(s) = app_{O'}(s)$ by Lemma 2.26. □

Note that acyclicity is a sufficient but not a necessary condition for a set of operators to be executable in some order, even for minimal disabling graphs. This is because the edges are independent of the state, exactly like the notion of interference in Definition 2.6. As in Example 2.14 two operators may form a cycle in the disabling graph but can nevertheless be executed in any order with the same results. However, for STRIPS operators and minimal disabling graphs acyclicity exactly coincides with executability in some order, as we show in Lemma 3.10. This fact was implicitly used in Theorem 2.28.

Lemma 3.10 *Let $\pi = \langle A, I, O, G\rangle$ be a problem instance and $\langle O, E\rangle$ a disabling graph for π such that $\langle o, o'\rangle \in E$ only if o affects o'. Let s be a state reachable from I by some sequence of operators in O and let $S = \{o_1, \ldots, o_n\}$ be a set of STRIPS operators so that $app_{o_1;\ldots;o_n}(s)$ and $app_S(s)$ are defined for some ordering $o_1, \ldots, o_n$ of S. Then the subgraph of $\langle O, E\rangle$ induced by S is acyclic.*

PROOF. Fact A: Since $app_S(s)$ is defined, there are no $\{\langle p, e, \emptyset\rangle, \langle p', e', \emptyset\rangle\} \subseteq S$ and $a \in A$ so that $a \in e$ and $\neg a \in e'$.

Since $app_{o_1;\ldots;o_n}(s)$ is defined, there are no $i \in \{1, \ldots, n-1\}$ and $j \in \{i+1, \ldots, n\}$ such that o_i affects o_j. If there were, o_i would make one of the literals in the precondition of o_j false and by Fact A no operator $o_k, k \in \{i+1, \ldots, j-1\}$ could make the precondition true again, and hence $app_{o_1;\ldots;o_j}(s)$ would not be defined. As no operator in S affects a later operator, and there is an edge from an operator to another only if the former affects the latter, the subgraph of $\langle O, E\rangle$ induced by S is acyclic. □

Next we discuss three ways of deriving constraints that guarantee that operators occupying one SCC of a disabling graph can be totally ordered to a valid plan.

[2]In $\forall$-step semantics simultaneous application is allowed only if the subgraph induced by all applied operators does not have *any* edges.

3.4.2 *Encoding of size $\mathcal{O}(n^3)$.* We can exactly test that the intersection of one SCC and a set of simultaneous operators do not form a cycle. The next encoding allows the maximum parallelism with respect to a given disabling graph, but it is expensive in terms of formula size.

We use auxiliary propositional variables $c_t^{i,j}$ for all operators with indices i and j indicating that the operators $o^i, o^1, o^2, \ldots, o^n, o^j$ are applied and each operator affects its immediate successor in the sequence. Let o^i and $o^{i'}$ belong to the same SCC of the disabling graph and let there be an edge from o^i to $o^{i'}$. Then we have the formulae $(o_t^i \wedge o_t^{i'}) \rightarrow c_t^{i,i'}$ and $(o_t^i \wedge c_t^{i',j}) \rightarrow c_t^{i,j}$ for all j such that $i' \neq j \neq i$. Further we have formulae $\neg(o_t^i \wedge c_t^{i',i})$ for preventing the completion of a cycle.

There is a cubic number of formulae, each having a constant size (two or three variable occurrences). The number of propositional variables $c_t^{i,j}$ is quadratic in the number of operators in an SCC. Some problems have SCCs of hundreds or thousands of operators, and this would mean millions or billions of formulae, which makes the encoding impractical.

Theorem 3.11 *Let $\pi = \langle A, I, O, G\rangle$ be a problem instance. There is a $\exists$-step plan of length l for π if $\Phi_{\pi,l} \wedge \Phi_{O,l}^{\exists step,1}$ is satisfiable.*

PROOF. Let v be a valuation such that $v \models \Phi_{\pi,l} \wedge \Phi_{O,l}^{\exists step,1}$. Define for all $i \in \{0, \ldots, l\}$ the state s_i as the valuation of A such that $s_i(a) = v(a_i)$ for every $a \in A$. Define $S_j = \{o \in O | v(o_j) = 1\}$ for all $j \in \{0, \ldots, l-1\}$. By Theorem 3.1 we only have to test the condition that for $\langle S_0, \ldots, S_{l-1}\rangle$, its execution $s_0, \ldots, s_l$ and every $i \in \{0, \ldots, l-1\}$ there is a total ordering $o_1, \ldots, o_n$ of S_i such that $app_{o_1;\ldots;o_n}(s_i) = app_{S_i}(s_i)$.

By Lemma 3.9 it suffices to show that the subgraph of the disabling graph induced by $S_i \cap C$ for every SCC C of the disabling graph is acyclic. For the sake of argument assume that the subgraph has a cycle. Hence there are operators $o'^1, \ldots, o'^m$ in S_i such that o'^j affects o'^{j+1} for all $j \in \{1, \ldots, m-1\}$ and o'^m affects o'^1. But the formulae

$$o_i'^{m-1} \wedge o_i'^m \rightarrow c_i^{m-1,m}, o_i'^{m-2} \wedge c_i^{m-1,m} \rightarrow c_i^{m-2,m}, \ldots, o_i'^1 \wedge c_i^{2,m} \rightarrow c_i^{1,m}, \neg(o_i'^m \wedge c_i^{1,m})$$

together with $o_i'^1, \ldots, o_i'^m$ are inconsistent. Since these formulae are conjuncts of $\Phi^{\exists step,1}$, there can be no cycle in the subgraph induced by $S_i \cap C$. □

3.4.3 *Encoding of size $\mathcal{O}(e \log_2 n)$.* A more compact encoding is obtained by assigning a $\log_2 n$-bit binary number to each of the n operators and by requiring that the number of operator o is lower than that of o' if there is an edge from o' to o in the disabling graph.[3] The size of the encoding is $\mathcal{O}(e \log_2 n)$ where e is the number of edges in the disabling graph and n is the number of operators.

For every operator o and time point t we introduce the propositional variables $i_t^{o,0}, \ldots i_t^{o,k}$ where $k = \lceil \log_2 n \rceil - 1$ for encoding o's index at time point t.

So, for any operators o and o' so that o' affects o use the following formula for guaranteeing that the edges are always from an operator with a higher index to a lower index.

$$(o_t \wedge o'_t) \rightarrow GT(i_t^{o',0}, \ldots, i_t^{o',k}; i_t^{o,0}, \ldots, i_t^{o,k}) \tag{13}$$

Above $GT(i_t^{o',0}, \ldots, i_t^{o',k}; i_t^{o,0}, \ldots, i_t^{o,k})$ is a formula comparing two k-bit binary numbers. There are such formulae that have a size that is linear in the number of bits.

[3]This encoding has also been independently discovered by Victor Khomenko [2005a].

Theorem 3.12 *Let $\pi = \langle A, I, O, G\rangle$ be a problem instance. There is a $\exists$-step plan of length l for π if $\Phi_{\pi,l} \wedge \Phi_{O,l}^{\exists step,2}$ is satisfiable.*

PROOF. Similarly to the proof of Theorem 3.11, we have to show that the subgraph induced by every set of simultaneous operators is acyclic. Formula 13 guarantees that the index of an operator to which there is an edge from another operator is lower than the index of the latter. The existence of a cycle would mean that there are also edges from an operator with a lower index to an operator to a higher index but as such edges do not exist, there are no cycles in the graph. □

Note that given a set of literals describing which operators are applied at a given time point, for the encoding in Section 3.4.2 unit resolution is sufficient for determining whether there is a cycle, but not for the encoding in Section 3.4.3.

3.4.4 *A linear-size encoding based on a fixed ordering of operators.* Our third encoding does not allow all the parallelism allowed by the preceding encodings but it leads to small formulae and seems to be very efficient in practice. With this encoding the set of formulae constraining parallel application *is a subset* of those for the less permissive $\forall$-step semantics. One therefore receives two benefits simultaneously: possibly much shorter parallel plans and formulae with a smaller size / time points ratio.

The idea is to impose beforehand an (arbitrary) ordering on the operators $o^1, \ldots, o^n$ in an SCC and to allow parallel application of two operators o^i and o^j such that o^i affects o^j only if $i \geq j$. Of course, this restriction to one fixed ordering may rule out many sets of parallel operators that could be applied simultaneously according to some other ordering than the fixed one.

A trivial implementation of this idea (similar to the $\forall$-step semantics encoding in Section 3.2.1) has a quadratic size because of the worst-case quadratic number of pairs of operators that may not be simultaneously applied. However, we may use one half of the implications in the linear-size encoding for $\forall$-step semantics from Section 3.2.2. The linear-size encoding for the constraints for $\exists$-step semantics is thus simply the conjunction of formulae

$$\textit{chain}(o^1, \ldots, o^n; E_m; R_m; m)$$

for every literal m where E_m is the set of operators that may falsify m ($\overline{m}$ occurs as an atomic effect) and R_m is the set of operators that may require m to remain true (m occurs in the antecedent of a conditional effect or positively in the precondition).

Theorem 3.13 *Let $\pi = \langle A, I, O, G\rangle$ be a problem instance. There is a $\exists$-step plan of length l for π if $\Phi_{\pi,l} \wedge \Phi_{O,l}^{\exists step,3}$ is satisfiable.*

PROOF. Let v be a valuation such that $v \models \Phi_{\pi,l} \wedge \Phi_{O,l}^{\exists step,3}$. Define for all $i \in \{0, \ldots, l\}$ the state s_i as the valuation of A such that $s_i(a) = v(a_i)$ for every $a \in A$. Define $S_j = \{o \in O | v(o_j) = 1\}$ for all $j \in \{0, \ldots, l-1\}$. Consider an SCC C of the disabling graph and the fixed ordering $o'^1, \ldots, o'^{n'}$ of the operators in $C \cap S_i$ for some $i \in \{0, \ldots, l-1\}$.

The formulae $\textit{chain}(o^1, \ldots, o^n; E_m; R_m; m)$ in $\Phi_{O,l}^{\exists step,3}$ for all literals m guarantee that if o'^j affects o'^k, then $k < j$. By Lemma 2.26 $\textit{app}_{S_i}(s_i) = \textit{app}_{o'^1;\ldots;o'^{n'}}(s_i)$. Hence the definition of $\exists$-step plans is satisfied. □

4. EXPERIMENTS

The shortest encodings of the three semantics in Sections 3.2.2, 3.3 and 3.4.4 have sizes that are linear in the size of the problem instance and the number of time points, and are therefore asymptotically optimal. The question arises whether the potentially much smaller number of time points makes $\exists$-step semantics more efficient than $\forall$-step semantics and whether the potentially much smaller number of plans makes the process semantics more efficient than $\forall$-step semantics. In this section we answer these questions by comparing the different encodings of the three semantics with respect to a number of planning problems.

We consider two problem classes. First, as a way of measuring the efficiency of the encodings on "average" problem instances, we sample problem instances from the space of all problem instances characterized by certain parameter values, following Bylander [1996] and Rintanen [2004b]. The problem instances we consider are rather small, 40 state variables and up to 280 operators, but rather challenging in the phase transition region.

Second, we consider some of the benchmarks used by the planning community. These problem instances have a simple interpretation in terms of real-world planning tasks, like simple forms of transportation planning. In contrast to the problem instances in the phase transition study, the numbers of state variables and operators in these problems are much higher (up to several thousands of state variables and tens of thousands of operators), and most of these problems can be solved rather easily by domain-specific polynomial time algorithms when no optimality criteria (for example minimal number of operators in a plan) have to be satisfied.

4.1 Implementation details

We briefly discuss details of the implementation of our translator from the planning domain description language PDDL [Ghallab, Howe, Knoblock, McDermott, Ram, Veloso, Weld, and Wilkins 1998] into propositional formulae in conjunctive normal form.

The planning domain description language PDDL allows describing schematic operators that are instantiated with a number of objects. For some of the standard benchmark problems the number of operators produced by a naïve instantiation procedure is astronomic, and indeed all practical planner implementations rely on heuristic techniques for avoiding the generation of ground operators that could never be part of a plan because no state satisfying the precondition of the operator can be reached.

After instantiating the schematic PDDL operators, we perform a simple polynomial-time reachability analysis for the possible values of state variables to identify operators that can never be applied. For example, in the 1998 and 2000 AIPS planning competition logistics problems there are operators for driving trucks between locations outside the truck's home city, but the truck can never leave its home city. Hence the state variables indicating that the truck's location is a non-home city can never be true. This analysis allows eliminating many irrelevant operators and it is similar to the reachability analysis performed by the GraphPlan [Blum and Furst 1997] and BLACKBOX [Kautz and Selman 1999] planners.

Similarly to BLACKBOX [Kautz and Selman 1999] and other implementations of satisfiability planning, our translation includes formulae $l_t \vee l'_t$ for invariants $l \vee l'$ as produced by the algorithm by Rintanen [1998]. This algorithm is defined for STRIPS operators only but can be generalized to arbitrary operators [Rintanen 2005].

In the experiments we use disabling graphs that are not necessarily minimal but can be computed in polynomial time. The test of whether two operators can be simultaneously applied in some state is not exact: we only test whether the unconditional effects contradict directly or through an invariant and whether the preconditions have conjuncts that are complementary literals or contradict through an invariant. For STRIPS operators the graphs are minimal whenever the invariants are sufficient for determining whether a state in which two operators are both executable is reachable.

The orderings in the $\exists$-step encoding of Section 3.4.4 were the ones in which the operators came out of our PDDL front-end. Better orderings that minimize the number of pairs of operators o and o' such that o precedes and affects o' could be produced by heuristic methods. They can potentially increase parallelism and improve runtimes.

The AIPS 2000 planning competition Schedule benchmarks contain conditional effects $m \rhd \overline{m}$, sometimes simultaneously with effects m. The purpose of this is apparently to make it difficult for planners like GraphPlan [Blum and Furst 1997] or BLACKBOX [Kautz and Selman 1999] to apply several operators in parallel. Replacing effects $m \rhd \overline{m}$ by preconditions $\overline{m}$ whenever also m is an unconditional effect and by effects $\overline{m}$ whenever m is not an unconditional nor a conditional effect of the operator, is a transformation that preserves the semantics of the operators exactly and for this benchmark allows much more parallelism. The front-end of our translator performs this transformation.

The SAT solvers we use only accept formulae in conjunctive normal form (CNF) as input. Therefore all the propositional formulae have to be transformed to CNF. We use a simple scheme for doing this. For any subformula of the form $(\phi_1 \wedge \phi_2) \vee \psi$ we introduce an auxiliary variable x, replace the subformula by $x \vee \psi$ and add $x \rightarrow \phi_1$ and $x \rightarrow \phi_2$ to our set of formulae. Note that almost all of the formulae in our encodings are already in CNF (modulo equivalences like $\neg(\phi \wedge \psi) \leftrightarrow \neg\phi \vee \neg\psi$.) Exceptions to this are the precondition axioms for operators with disjunctive preconditions and effect axioms for operators with conditional effects.

For effect axioms $o_t \rightarrow e_t$ we only include those effects in e that are not consequences of other effects and invariants. For example, many operators in the standard benchmarks have effects *at(A,L1)*$\wedge\neg$*at(A,L2)* for representing the movement of an object from location 2 to location 1. Then $\neg$*at(A,L1)*$\vee\neg$*at(A,L2)* is an invariant that is included in the problem encoding. Since $\neg$*at(A,L2)* is a consequence of the invariant together with *at(A,L1)*, the effect axiom 2 does not have to state this explicitly. This reduces the size of the formulae slightly and has a small effect on runtimes.

4.2 Experimental setting

For the experiments we use a 3.6 GHz Intel Xeon processor with 512 KB internal cache and the Siege SAT solver version 4 by Ryan of the Simon Fraser University.

In addition to Siege V4, we ran tests with the May 13, 2004 version of zChaff. The runtimes are close to the ones for Siege, often worse but in some cases slightly better. We could solve some of the biggest structured instances (Section 4.4) in a reasonable time only with Siege. BerkMin and some of the best solvers in the 2005 SAT solver competition are also rather good on the planning problems.

As Siege V4 uses randomization, its runtimes vary, in some cases considerably. For the structured problems the tables give the average runtimes over 100 runs and 95 percent confidence intervals for the average runtimes. As it is not known what the distribution of Siege runtimes on a given instance is, we calculate the confidence intervals by using a standard bootstrapping procedure [Efron and Tibshirani 1986; Efron and Tibshirani 1993]. From the sample of 100 runtimes we resample (with replacement) 4000 times a sample of

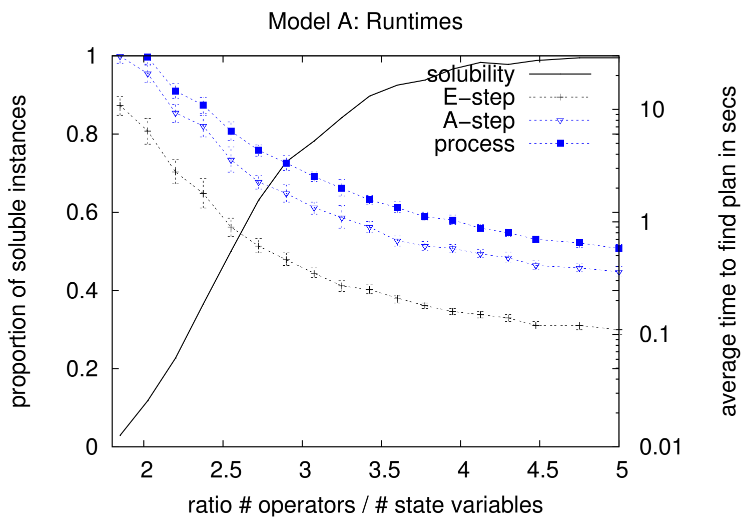


Fig. 3. Runtimes of ∃-step, ∀-step and process semantics on problem instances with 40 state variables sampled from the phase transition region.

100 runtimes and then look at the distribution of these averages. The 95 percent confidence interval is obtained as the 2.5 and 97.5 percentiles of this distribution.

4.3 Problem instances sampled from the phase transition region

We considered problem instances with $|A| = 40$ state variables, corresponding to state spaces with $2^{40} \sim 10^{12}$ states, and STRIPS operators with 3 literals in the preconditions and 2 literals in the effect, following Model A of Rintanen [2004b] in which precondition literals are chosen randomly and independently, and effect literals are chosen randomly so that each propositional variable has about the same number of occurrences in an atomic effect, both negatively and positively. In the initial state all state variables are false and in the unique goal state all state variables are true. Problem instances of this size are very hard for existing planning algorithms; see further discussion in Section 6.3. We generated about 1000 soluble problem instances for ratios $\frac{|O|}{|A|}$ of operators to state variables varying from 1.85 to 5 at an interval of about 0.3. The number of operators then varied from 74 to 280. To find 1000 soluble instances for the smaller ratios we had to generate up to 45000 instances most of which are insoluble. Since we did not have a complete insolubility test, we do not know how many of the instances that we could not solve within our limits on plan length (60 time points) and CPU time (3 minutes per formula) are really insoluble. For the ∀-step and the process semantics the number of instances solved within the time limit was slightly smaller than for the ∃-step semantics, so the actual performance difference is slightly bigger than what the diagram suggests.

Figure 3 depicts the average runtimes of Siege with the ∃-step (the linear-size encoding from Section 3.4.4), ∀-step (the linear-size encoding from Section 3.2.2) and process semantics (the linear-size encoding from Section 3.3 based on the linear-size ∀-step encoding from Section 3.2.2). There are two sources of imprecision in the runtime comparison, the variation of runtimes of Siege due to randomization and the random variation in the

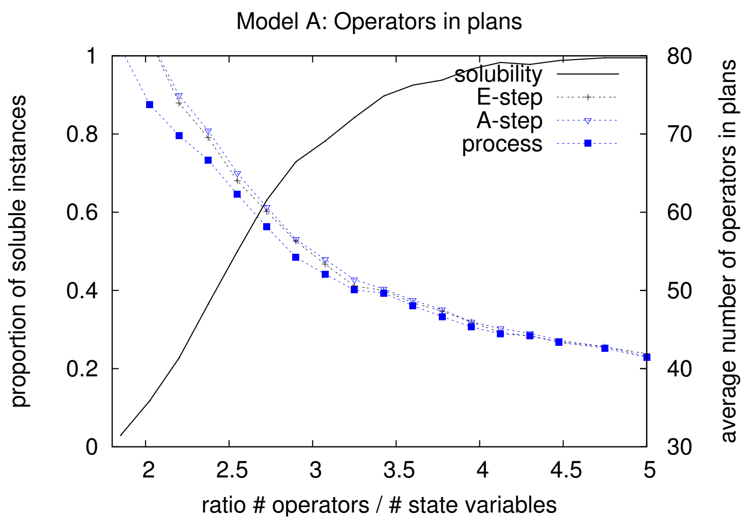


Fig. 4. Numbers of operators in plans for ∃-step, ∀-step and process semantics on problem instances with 40 state variables sampled from the phase transition region.

properties of problem instances sampled from the space of all problem instances. For this reason we give estimates on the accuracy of the averages of runtimes. The diagrams depicting the runtimes give error bars indicating the 95 percent confidence intervals for the runtimes. Note that the scale of the runtime diagram is logarithmic.

Figure 4 depicts the average numbers of operators in the plans. Figure 5 depicts the average number of time points in the plans. The process and ∀-step semantics share the curve because the shortest number of time points of a plan for any problem instance is the same for both.

As is apparent from the diagrams, the ∃-step semantics is by far the most efficient of the three. The efficiency is directly related to the fact that with ∃-step semantics the shortest plans often have less time points than with the ∀-step and process semantics. The encoding for the process semantics is the slowest, most likely because of the higher number of propositional variables and clauses and the ineffectiveness of the process constraints on these problems.

Interestingly, the number of operators in the ∃-step and ∀-step plans is almost exactly the same despite the fact that the ∀-step semantics needs more time points. On the other hand, process semantics imposes stricter constraints on the plans than the ∀-step semantics, and the number of operators is therefore slightly smaller.

4.4 Structured problem instances

We evaluate the different semantics on a number of benchmarks from the AIPS planning competitions of years 1998, 2000 and 2002. For a discussion of these benchmarks and their properties see Section 5.5. We also tried the benchmarks from the year 2004 competition, but, although most of them are easy to solve, they result in very big formulae, and the relative behavior of the encodings of the different semantics on them is similar to the benchmarks we report in this paper. Hence we did not run exhaustive tests with them.

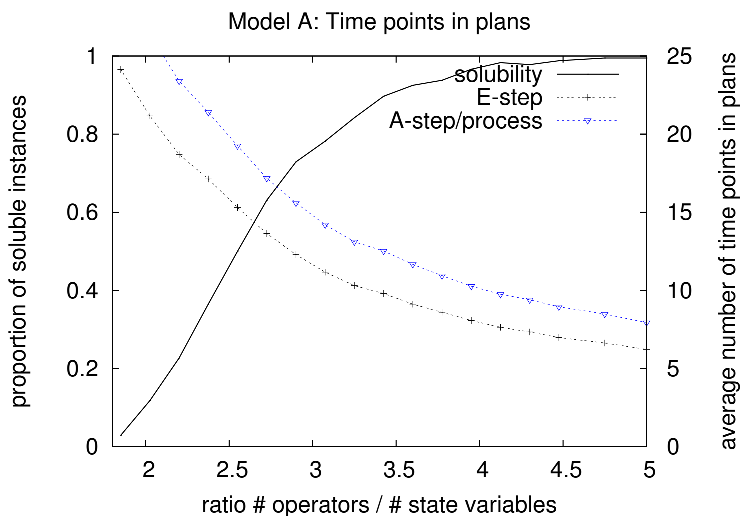


Fig. 5. Numbers of time points in plans for ∃-step, ∀-step and process semantics on problem instances with 40 state variables sampled from the phase transition region. For ∀-step and process semantics the number of time points is always the same.

For all other benchmarks we use the STRIPS version, but for the Schedule benchmark we use the ADL version because with the STRIPS version our translator has problems with the very high number of operators. However, the simplification mentioned in Section 4.1 transforms also these operators to STRIPS operators.

In Tables I, X, XI, XII, XIII XIV, XV, XVI and XVII (all but the first are in the appendix) we present for each problem instance the runtimes for the formulae corresponding to the highest number of time points without a plan (truth value F) and the first satisfiable formula corresponding to a plan (truth value T). The rows marked with the question mark indicate that none of the runs successfully terminated and we therefore do not know whether the formulae are satisfiable or unsatisfiable. The column ∃-*step* is for the ∃-step semantics encoding in Section 3.4.4, the column *process* for the process semantics encoding in Section 3.3, the column ∀-*step* for the worst-case quadratic ∀-step semantics encoding in Section 3.2.1, and the column ∀-*step l.* for the linear ∀-step semantics encoding in Section 3.2.2.

Runtimes for ∃-step semantics are in most cases reported on their own lines because its shortest plan lengths differ from the other semantics. Each runtime is followed by the upper and lower bounds of the 95 percent confidence intervals. We indicate by a dash - the formulae for which not all runs finished within 180 seconds.

In Table II we compare the semantics in terms of the number of operators in plans. Blocks World problems are sequential (only one operator can be applied at a time) and numbers of operators equal numbers of time points. The average number of operators is followed by the lowest and the highest number of operators any plan we found had.

In Table III we present data on formula sizes.

4.4.1 ∃-*step semantics vs.* ∀-*step semantics.* The lowest runtimes are usually obtained with the ∃-step semantics. It is often one or two orders of magnitude faster. For problem

instance	len	val	∃-step	process	∀-step	∀-step l.
log-16-0	7	F	0.01 $^{0.01}_{0.01}$			
log-16-0	8	T	0.03 $^{0.03}_{0.04}$			
log-16-0	12	F		0.62 $^{0.57}_{0.67}$	0.30 $^{0.27}_{0.33}$	0.79 $^{0.73}_{0.86}$
log-16-0	13	T		7.46 $^{6.96}_{7.98}$	1.35 $^{1.19}_{1.52}$	2.27 $^{2.04}_{2.50}$
log-17-0	8	F	0.15 $^{0.14}_{0.15}$			
log-17-0	9	T	0.02 $^{0.02}_{0.02}$			
log-17-0	13	F		3.06 $^{2.93}_{3.19}$	1.97 $^{1.89}_{2.05}$	2.25 $^{2.15}_{2.35}$
log-17-0	14	T		14.40 $^{13.71}_{15.11}$	3.22 $^{2.93}_{3.55}$	4.48 $^{4.07}_{4.91}$
log-18-0	8	F	0.13 $^{0.13}_{0.14}$			
log-18-0	9	T	0.33 $^{0.26}_{0.40}$			
log-18-0	14	F		8.18 $^{7.74}_{8.67}$	5.83 $^{5.50}_{6.17}$	6.77 $^{6.47}_{7.08}$
log-18-0	15	T		–	7.84 $^{6.78}_{9.09}$	14.95 $^{13.13}_{16.78}$
log-19-0	8	F	0.23 $^{0.22}_{0.25}$			
log-19-0	9	T	0.33 $^{0.22}_{0.46}$			
log-19-0	14	F		10.23 $^{9.54}_{10.95}$	11.22 $^{10.47}_{11.98}$	13.39 $^{12.68}_{14.12}$
log-19-0	15	T		–	29.10 $^{25.33}_{33.05}$	–
log-20-0	8	F	0.25 $^{0.24}_{0.26}$			
log-20-0	9	T	0.88 $^{0.64}_{1.17}$			
log-20-0	14	F		12.30 $^{11.76}_{12.83}$	10.63 $^{9.96}_{11.32}$	12.01 $^{11.36}_{12.67}$
log-20-0	15	T		–	–	41.17 $^{36.81}_{45.80}$

Table I. Runtimes of Logistics problems

instance	len	∃-step	process	∀-step	∀-step l.
depot-16-4398	8	53.00 $^{53}_{53}$	43.22 $^{36}_{48}$	43.40 $^{38}_{50}$	38.97 $^{35}_{44}$
driver-4-4-8	9	54.19 $^{50}_{61}$			
driver-4-4-8	11		55.45 $^{50}_{61}$	52.32 $^{50}_{58}$	51.47 $^{50}_{58}$
gripper-3	8	23.21 $^{23}_{24}$			
gripper-3	15		23.00 $^{23}_{23}$	23.00 $^{23}_{23}$	23.00 $^{23}_{23}$
log-16-0	8	122.74 $^{105}_{131}$			
log-16-0	13		146.47 $^{125}_{167}$	123.91 $^{106}_{141}$	125.32 $^{108}_{143}$
freecell5-4	13	32.99 $^{30}_{35}$	32.65 $^{30}_{33}$	34.02 $^{31}_{35}$	32.46 $^{30}_{33}$
elev-str-f24	17	58.38 $^{51}_{63}$			
elev-str-f24	32		40.00 $^{40}_{40}$	40.00 $^{40}_{40}$	40.00 $^{40}_{40}$
satel-17	4	191.55 $^{82}_{274}$			
satel-17	6		95.00 $^{83}_{106}$	122.14 $^{92}_{158}$	96.73 $^{85}_{105}$
sched-30-0	11	43.88 $^{40}_{50}$	50.79 $^{45}_{53}$	45.06 $^{39}_{53}$	41.63 $^{38}_{46}$
zeno-5-10	4	34.36 $^{34}_{35}$			
zeno-5-10	6		43.32 $^{38}_{48}$	46.80 $^{38}_{58}$	40.63 $^{35}_{46}$

Table II. Numbers of operators in plans

instance	len	∃-step			process			∀-step			∀-step l.		
		$\frac{P}{10^3}$	$\frac{C}{10^3}$	MB	$\frac{P}{10^3}$	$\frac{C}{10^3}$	MB	$\frac{P}{10^3}$	$\frac{C}{10^3}$	MB	$\frac{P}{10^3}$	$\frac{C}{10^3}$	MB
block-18-0	58	58.9	696.8	10.9	264.9	1218.8	24.9	58.9	696.8	10.9	201.0	1120.1	18.9
block-20-0	60	74.9	937.8	14.8	338.4	1607.2	32.9	74.9	937.8	14.8	257.7	1482.0	25.2
block-22-0	72	108.3	1431.0	22.9	490.5	2406.9	49.5	108.3	1431.0	22.9	374.6	2225.5	38.4
depot-17-6587	7	24.1	256.3	3.9	154.7	611.6	12.8	24.1	269.8	4.1	144.2	586.2	9.7
depot-18-1916	12	75.7	864.9	13.7	484.2	2052.0	45.4	75.7	899.4	14.2	457.7	1968.3	34.2
depot-15-4534	20	93.0	882.8	14.5	594.8	2360.2	53.7	93.0	918.8	15.0	550.2	2243.9	39.4
driver-2-3-6e	12	25.4	110.6	1.6	66.2	206.4	3.7	21.5	157.0	2.3	42.9	174.1	2.6
driver-3-3-6b	11	22.3	93.4	1.4	54.9	178.2	3.3	18.0	144.9	2.1	39.2	153.5	2.3
driver-4-4-8	11	48.5	210.7	3.3	117.0	401.3	7.7	37.7	382.5	5.8	89.4	352.3	5.4
gripper-2	11	1.0	4.7	0.1	2.9	8.8	0.1	1.0	5.3	0.1	1.5	7.2	0.1
gripper-3	15	1.8	8.7	0.1	5.0	16.1	0.3	1.8	9.7	0.1	2.7	13.2	0.2
gripper-4	17	2.4	12.5	0.2	6.9	23.1	0.4	2.4	13.9	0.2	3.7	19.0	0.3
log-16-0	13	18.7	105.4	1.5	46.6	174.3	3.1	18.7	139.1	2.0	27.0	146.3	2.2
log-20-0	15	29.1	174.8	2.5	72.4	284.6	5.1	29.1	236.6	3.5	42.5	240.6	3.6
log-24-0	15	37.8	240.8	3.5	94.3	385.1	6.9	37.8	333.0	4.9	55.9	328.2	5.0
elev/str-f8	12	1.0	2.4	0.0	4.1	8.1	0.1	1.0	3.0	0.0	2.1	5.7	0.1
elev/str-f12	14	2.4	5.8	0.1	10.3	21.4	0.3	2.4	7.7	0.1	5.7	15.6	0.2
elev/str-f16	22	6.4	15.7	0.2	27.8	60.7	1.1	6.4	21.0	0.3	16.2	44.7	0.7
elev/str-f20	26	11.5	28.4	0.4	50.5	112.5	2.0	11.5	38.3	0.6	30.2	83.7	1.3
elev/str-f24	28	17.5	43.4	0.7	77.5	174.7	3.1	17.5	58.9	0.9	47.2	131.0	2.0
satel-14	8	37.7	129.6	2.0	108.1	347.0	6.7	37.7	267.0	4.1	98.5	309.4	4.8
satel-15	8	49.0	168.5	2.7	142.0	454.0	9.2	49.0	327.3	5.1	130.1	405.3	6.6
satel-16	6	46.8	161.5	2.6	136.6	430.1	8.6	46.8	333.7	5.2	125.7	386.3	6.3
satel-17	6	54.0	185.6	3.0	160.6	500.1	10.1	54.0	346.7	5.4	148.5	449.8	7.5
satel-18	8	31.7	108.5	1.7	91.3	290.2	5.5	31.7	221.1	3.4	82.4	258.1	4.0
sched-10-0	7	7.3	40.2	0.6	16.5	58.4	1.1	3.4	73.5	1.0	11.3	53.0	0.8
sched-20-0	9	18.2	101.5	1.6	40.7	148.4	3.0	8.4	285.0	3.9	28.5	134.8	2.1
sched-30-0	11	32.9	185.3	3.0	72.9	271.7	5.5	15.1	700.0	10.3	51.4	246.6	3.9
sched-40-0	15	58.8	334.3	5.4	129.6	492.4	10.9	27.0	1595.3	24.6	91.8	445.9	7.1
sched-50-0	17	82.7	480.3	7.8	182.0	704.7	15.9	38.0	2720.7	42.0	129.2	638.5	10.9
zeno-3-8b	6	9.1	49.0	0.7	42.0	139.4	2.6	9.1	144.1	2.0	39.5	130.7	2.0
zeno-5-10	6	39.2	220.8	3.6	195.2	653.8	13.9	39.2	814.8	12.3	190.1	618.9	10.5
zeno-5-15	6	59.0	332.7	5.5	291.0	979.0	21.1	59.0	1639.5	25.0	283.6	926.6	16.0
zeno-5-15b	6	78.0	309.5	5.5	391.9	1182.9	26.3	78.0	2111.3	32.6	383.6	1114.4	19.5

Table III. Sizes of formulae under the different encodings. The column $\frac{P}{10^3}$ gives the number of propositional variables in thousands, the column $\frac{C}{10^3}$ the number of clauses in thousands, and the column MB the size of the DIMACS encoded formulae in CNF in megabytes. The data are on the satisfiable formulae corresponding to the length of shortest existing plans under ∀-step semantics. The shortest ∃-step plans are in many cases shorter, and the required formulae then correspondingly smaller.

instances that are more difficult than those depicted in the tables the runtime differences are still bigger. Most of the benchmark problems allow parallelism, and in most of these cases ∃-step semantics allows more operators in parallel than the ∀-step semantics. For example in many of the problems involving transportation of objects by vehicles, with ∃-step semantics a vehicle can leave a location simultaneously with loading or unloading an object to or from the vehicle. The smaller parallel plan lengths directly lead to much faster planning.

For the Schedule benchmark ∃-step semantics does not allow more parallelism than the ∀-step semantics. The linear-size ∃-step semantics is as efficient as the linear-size ∀-step semantics encoding and slightly less efficient than the quadratic-size ∀-step semantics encoding as far as the unsatisfiable formulae are concerned. Interestingly, the relative efficiency of the encodings reverses for satisfiable formulae corresponding to plans. As shown in Table IV, for satisfiable formulae the SAT solver runtimes more closely reflect the relative sizes of the encodings: the linear-size ∃-step encoding is the fastest, followed by the linear-size ∀-step encoding and the quadratic size ∀-step encoding.

Numbers of operators in plans for the different encodings do not seem to follow any regular pattern. In same cases the process semantics plans have the most operators, in others the ∀-step or the ∃-step plans.

4.4.2 *Process semantics vs. ∀-step semantics.* Contrary to our expectations based on the earlier results by Heljanko [2001] on Petri net deadlock detection problems, process semantics does not provide an advantage over ∀-step semantics on these problems although there are often far fewer potential plans to consider. When showing the inexistence of plans of certain length, the additional constraints could provide a big advantage similarly to symmetry-breaking constraints.

Differences to the results by Heljanko are likely to be because of differences between the application area and the type of SAT solvers and encodings used. The problem with the planning problems would appear to be the high number of long clauses that usually do not lead to pruning the search space and just add an overhead. In an earlier paper we rejected full process semantics and only considered clauses with a length below a small constant [Rintanen, Heljanko, and Niemelä 2004]. In some cases the constraints substantially improved runtimes, but in most cases there was no effect because of the very small number of additional short clauses.

4.4.3 *Linear vs. quadratic ∀-step encoding.* It is interesting to make a comparison between the quadratic and linear size encodings of the ∀-step semantics constraints respectively discussed in Sections 3.2.1 and 3.2.2. Even though the worst-case formula sizes are smaller with the linear encoding, this did not directly translate into smaller formulae and improved runtimes. First of all, even though the encoding from Section 3.2.1 is worst-case quadratic, the number of clauses $\neg o \vee \neg o'$ is often small because not all pairs of operators interfere. Also many pairs of interfering operators cannot be simultaneously applied, and hence the corresponding clauses are not needed.

The only benchmark series in which the linear-size encoding substantially improves on the worst-case quadratic-size encoding is Schedule. This is because in this benchmark there is a very high number of pairs of interfering operators that can be applied simultaneously, and the quadraticity therefore very clearly shows up. Hence the linear-size encoding leads to much smaller formulae. Better runtimes are however obtained only for plan lengths higher than the shortest existing plans, as shown in Figure IV. On still bigger instances the

instance	len	val	∃-step	∀-step	∀-step l.
sched-35-0	13	T	3.43 $^{2.65}_{4.33}$	3.86 $^{3.13}_{4.65}$	3.14 $^{2.46}_{3.95}$
sched-35-0	14	T	2.10 $^{1.78}_{2.44}$	3.08 $^{2.63}_{3.62}$	1.63 $^{1.37}_{1.90}$
sched-35-0	15	T	1.39 $^{1.20}_{1.57}$	2.81 $^{2.41}_{3.26}$	1.83 $^{1.58}_{2.13}$
sched-35-0	16	T	1.41 $^{1.22}_{1.62}$	2.30 $^{1.99}_{2.65}$	1.43 $^{1.22}_{1.69}$
sched-35-0	17	T	1.28 $^{1.13}_{1.43}$	3.08 $^{2.66}_{3.53}$	1.43 $^{1.24}_{1.63}$
sched-35-0	18	T	1.22 $^{1.07}_{1.37}$	3.95 $^{3.28}_{4.82}$	1.52 $^{1.26}_{1.86}$
sched-35-0	19	T	1.20 $^{1.04}_{1.37}$	5.62 $^{4.73}_{6.56}$	1.40 $^{1.25}_{1.54}$
sched-35-0	20	T	1.31 $^{1.17}_{1.46}$	4.77 $^{4.18}_{5.39}$	1.41 $^{1.24}_{1.59}$
sched-35-0	21	T	1.04 $^{0.90}_{1.19}$	4.80 $^{4.26}_{5.36}$	1.07 $^{0.93}_{1.24}$
sched-35-0	22	T	1.37 $^{1.20}_{1.58}$	14.97 $^{13.44}_{16.58}$	1.38 $^{1.23}_{1.54}$
sched-35-0	23	T	1.16 $^{1.02}_{1.31}$	6.17 $^{5.36}_{7.05}$	1.26 $^{1.10}_{1.44}$
sched-35-0	24	T	1.64 $^{1.44}_{1.83}$	10.14 $^{8.89}_{11.51}$	2.13 $^{1.85}_{2.42}$
sched-35-0	25	T	1.68 $^{1.47}_{1.90}$	20.52 $^{18.31}_{22.69}$	1.83 $^{1.58}_{2.12}$
sched-35-0	26	T	1.54 $^{1.37}_{1.71}$	17.65 $^{15.64}_{19.71}$	2.11 $^{1.82}_{2.42}$
sched-35-0	27	T	1.77 $^{1.53}_{2.02}$	13.46 $^{11.74}_{15.36}$	1.56 $^{1.34}_{1.80}$
sched-35-0	28	T	1.56 $^{1.38}_{1.76}$	22.96 $^{20.09}_{26.10}$	2.22 $^{1.86}_{2.64}$

Table IV. Runtimes for the satisfiable formulae for different plan lengths

instance	SCCs							
block-34-0	2312×1							
depot-22-1817	22252×1							
gripper-5	98×1							
elev/str-f60	3600×1							
log-41-0	7812×1							
satel-20	4437×1							
zeno-5-25b	31570×1							
driver-4-4-8	16×10	16×9	32×8	48×7	16×6	32×5	32×4	1312×1
sched-51-0	1×1173	1×51	1×1					
freecell8-4	1×6882	99×1						

Table V. Sizes of SCCs of Disabling Graphs: $n \times m$ means that there are n SCCs of size m.

differences are still more pronounced. These differences between the linear and quadratic size encodings often mean much bigger differences in total runtimes on planners that use more sophisticated evaluation algorithms than the standard sequential one, for example the algorithm we consider in Section 5.3.

4.4.4 *Sizes of strong components of disabling graphs.* Some of the sizes of SCCs of disabling graphs are depicted in Table V. We only give the SCC sizes for one instance of each benchmark series because the SCC sizes for all instances of each series are similar. For example, all SCCs of all instances of the Blocks World, Depot, Gripper, Elevator, Logistics, Satellite and ZenoTravel have size 1. For the other benchmarks, the SCC sizes are a function of some of the problem parameters, like the number of vehicles.

Only few or no constraints on parallel operators are needed if all the strong components of the disabling graphs are small. This directly contributes to the small size of the formulae for the ∃-step semantics. However, it is not clear whether this per se is a reason for the efficiency of ∃-step semantics. For problems in which shortest ∃-step and shortest ∀-step plans have the same length, for example the blocks world problems, ∃-step encoding is not more efficient than the corresponding ∀-step semantics encoding.

instance	len	val	∀-step	blackbox
block-12-1	33	F	0.06 $^{0.06}_{0.06}$	0.20 $^{0.20}_{0.21}$
block-12-1	34	T	0.05 $^{0.05}_{0.05}$	0.22 $^{0.21}_{0.23}$
block-14-1	35	F	0.35 $^{0.34}_{0.35}$	18.02 $^{16.91}_{19.29}$
block-14-1	36	T	0.12 $^{0.11}_{0.14}$	5.65 $^{4.97}_{6.39}$
block-16-1	53	F	0.65 $^{0.63}_{0.68}$	33.38 $^{31.10}_{35.66}$
block-16-1	54	T	0.38 $^{0.36}_{0.40}$	13.85 $^{12.52}_{15.25}$
block-18-0	57	F	2.29 $^{2.22}_{2.36}$	–
block-18-0	58	T	1.07 $^{0.98}_{1.17}$	24.15 $^{21.61}_{26.90}$
log-17-0	13	F	1.97 $^{1.89}_{2.05}$	0.42 $^{0.40}_{0.44}$
log-17-0	14	T	3.22 $^{2.91}_{3.56}$	1.06 $^{0.91}_{1.22}$
log-18-0	14	F	5.83 $^{5.50}_{6.18}$	3.25 $^{2.98}_{3.55}$
log-18-0	15	T	7.84 $^{6.74}_{9.07}$	2.21 $^{1.86}_{2.59}$
log-19-0	14	F	11.22 $^{10.52}_{12.01}$	4.55 $^{4.30}_{4.82}$
log-19-0	15	T	29.10 $^{25.25}_{33.00}$	13.74 $^{11.99}_{15.53}$
log-20-0	14	F	10.63 $^{9.96}_{11.36}$	7.88 $^{7.52}_{8.26}$
log-20-0	15	T	–	15.94 $^{13.97}_{18.01}$
depot-14-7654	11	F	1.41 $^{1.34}_{1.48}$	0.30 $^{0.28}_{0.31}$
depot-14-7654	12	T	3.48 $^{3.19}_{3.78}$	1.17 $^{1.06}_{1.29}$
depot-16-4398	7	F	0.01 $^{0.01}_{0.01}$	0.01 $^{0.01}_{0.01}$
depot-16-4398	8	T	0.07 $^{0.06}_{0.07}$	0.01 $^{0.01}_{0.01}$
depot-18-1916	11	F	0.17 $^{0.16}_{0.17}$	28.41 $^{23.75}_{33.44}$
depot-18-1916	12	T	–	–
driver-2-3-6d	15	F	19.09 $^{18.22}_{19.98}$	43.44 $^{40.04}_{46.98}$
driver-2-3-6d	16	T	8.04 $^{7.16}_{9.00}$	18.94 $^{17.72}_{20.16}$
driver-2-3-6e	11	F	1.13 $^{1.07}_{1.19}$	0.60 $^{0.56}_{0.63}$
driver-2-3-6e	12	T	1.27 $^{1.10}_{1.45}$	1.51 $^{1.28}_{1.73}$
driver-3-3-6b	10	F	0.82 $^{0.77}_{0.87}$	0.60 $^{0.56}_{0.65}$
driver-3-3-6b	11	T	1.07 $^{0.90}_{1.25}$	0.76 $^{0.64}_{0.89}$
driver-4-4-8	10	F	1.30 $^{1.26}_{1.33}$	0.56 $^{0.52}_{0.61}$
driver-4-4-8	11	T	5.92 $^{5.34}_{6.50}$	19.35 $^{17.88}_{20.89}$
gripper-2	10	F	0.08 $^{0.08}_{0.09}$	0.34 $^{0.32}_{0.36}$
gripper-2	11	T	0.02 $^{0.01}_{0.02}$	0.12 $^{0.10}_{0.15}$
gripper-3	14	F	3.91 $^{3.49}_{4.35}$	41.07 $^{36.15}_{45.84}$
gripper-3	15	T	0.32 $^{0.19}_{0.46}$	2.82 $^{2.08}_{3.63}$

Table VI. Runtimes of the quadratic ∀-step semantics encoding vs. the BLACKBOX encoding

4.4.5 *Quadratic ∀-step encoding vs. the BLACKBOX encoding.* The BLACKBOX planner by Kautz and Selman [1999] is the best-known planner that implements the planning as satisfiability paradigm[4]. Our quadratic encoding of the ∀-step semantics (Section 3.2.1) is closest to the planning graph based encoding used in the BLACKBOX planner. We give a comparison between the runtimes for our quadratic ∀-step semantics encoding and the encoding used by BLACKBOX in Table VI,[5] and between the formula sizes in Table VII.

[4]Surprisingly, the SAT encodings of planning by the SATPLAN04 planner of Kautz et al. (unpublished work) which participated in the 2004 planning competition are for many benchmark problems much slower than the BLACKBOX encodings, and only in few cases it is somewhat faster.

[5]We were not able to test all the benchmarks with BLACKBOX because of certain bugs in BLACKBOX.

instance	len	∀-step			blackbox		
		$\frac{P}{10^3}$	$\frac{C}{10^3}$	MB	$\frac{P}{10^3}$	$\frac{C}{10^3}$	MB
block-12-1	34	15.71	152.4	2.23	13.12	1035.3	14.80
block-14-1	36	22.44	233.7	3.47	24.88	2938.5	44.59
block-16-1	54	43.54	485.1	7.51	42.73	6012.7	94.72
block-18-0	58	58.88	696.8	10.92	61.79	11091.9	176.58
log-17-0	14	20.12	149.8	2.16	10.41	431.8	6.15
log-19-0	15	29.08	236.6	3.46	15.47	897.1	12.98
log-21-0	16	30.98	252.3	3.70	19.97	1301.6	19.43
log-23-0	16	40.22	355.1	5.29	24.13	1973.5	29.97
log-25-0	15	56.66	556.3	8.42	28.70	3419.9	52.70
depot-14-7654	12	30.99	357.5	5.52	12.79	1952.6	27.96
depot-16-4398	8	13.72	143.5	2.10	4.12	237.7	3.33
depot-18-1916	12	75.67	899.4	14.18	33.42	14599.4	230.82
driver-2-3-6d	16	23.00	168.5	2.51	15.60	1809.6	26.44
driver-2-3-6e	12	21.52	157.0	2.32	11.45	1432.2	20.47
driver-3-3-6b	11	17.97	144.9	2.12	8.86	972.9	13.87
driver-4-4-8	11	37.73	382.5	5.81	15.54	3406.7	49.92
gripper-2	11	1.01	5.3	0.06	1.15	15.2	0.19
gripper-3	15	1.76	9.7	0.12	2.13	36.7	0.48
gripper-4	19	2.72	15.5	0.20	3.39	71.6	0.97

Table VII. Formula sizes of the quadratic ∀-step semantics encodings vs. the BLACKBOX encoding

The planning graph [Blum and Furst 1997] is a data structure that represents constraints $\neg o_t \vee \neg o'_t$ for pairs of interfering operators, 2-literal invariants, as well as 1-literal and 2-literal clauses that indicate that certain values of state variables and application of certain operators are not possible at given time points. The 2-literal clauses in planning graphs are called *mutexes*. A peculiarity of planning graphs is the NO-OP operators which are used as a marker for the fact that a given state variable does not change its value. The problem encoding used by BLACKBOX is based on translating the contents of planning graphs into 1-literal and 2-literal clauses.

For some of the easiest problems the BLACKBOX encoding is more efficient than the quadratic ∀-step semantics encoding (the Logistics problems and some instances of the Depot problem), but in many cases it is much less efficient, most notably on the Blocks World, Driver and Gripper problems. We believe that BLACKBOX's efficiency on the easier problems is due to the explicit reachability information in the planning graph that with our ∀-step semantics encoding has to be inferred, and the inefficiency in general is due to the bigger formulae BLACKBOX produces.

The BLACKBOX encoding results in much bigger formulae than the quadratic ∀-step encoding, for the biggest instances by factors up to 25. The main reason for this is the very straightforward translation of planning graphs into propositional formulae BLACKBOX uses. This includes many redundant interference mutexes for operators that can also be otherwise inferred not to be simultaneously executable as well as many mutexes between NO-OPs and operators.

The ∀-step semantics formulae often have almost twice as many propositional variables as the BLACKBOX formulae. This is due to the reachability information in the planning graphs that allows to infer that only certain operators are executable and that only certain state variable values are possible at some of the early time points. Roughly the same reduction could be obtained for our $\forall$-step semantics formulae by performing unit resolution and then eliminating all occurrences of propositional variables occurring in a unit clause by subsumption.

We conclude that the BLACKBOX encoding is roughly comparable to our quadratic encoding for the $\forall$-step semantics and hence in many cases much less efficient than our encoding for the $\exists$-step semantics. Further, the formulae for the BLACKBOX encoding are often several times bigger.

5. EVALUATION ALGORITHMS FOR PLANNING AS SATISFIABILITY

Earlier research on classical planning that splits plan search into finding plans of given fixed lengths, for instance the GraphPlan algorithm [Blum and Furst 1997] and planning as satisfiability [Kautz and Selman 1996] and related approaches [Rintanen 1998; Kautz and Walser 1999; Wolfman and Weld 1999; van Beek and Chen 1999; Do and Kambhampati 2001], have without exception adopted a sequential strategy for plan search. This strategy starts with (parallel) plan length 0, and if no such plans exist, continues with length 1, length 2, and so on, until a plan is found.

This standard sequential strategy is guaranteed to find a plan with the minimal number of time points. If only one operator is applied at every time point then the plans are also guaranteed to contain the minimal number of operators.

It seems that for finding a plan with the minimal number of time points the sequential strategy cannot in general be improved. For example, a strategy that increases the plan length by more than one until a satisfiable formula is found and then performs a binary search to find the shortest plan does not typically improve runtimes because the cost of evaluating the unsatisfiable formulae usually increases exponentially as the plan length increases.

However, when the objective is to find any plan, not necessarily with the minimal number of time points, we can use more efficient search strategies for plan search. The standard sequential strategy is often inefficient because the satisfiability tests for the last unsatisfiable formulae are often much more expensive than for the first satisfiable formulae. Consider the diagrams in Figures 6 and 7 that represent some standard benchmarks problems as well as the diagrams in Figure 8 that represent two difficult problem instances with 20 state variable sampled from the phase transition region [Rintanen 2004b]. Each diagram shows the cost of detecting the satisfiability or unsatisfiability of formulae that represent the existence of plans of different lengths. Black bars depict unsatisfiable formulae and grey bars satisfiable formulae.

When the plan quality (the number of time points) is not a concern, we would like to run a satisfiability algorithm with the satisfiable formula for which the runtime of the algorithm is the lowest. Of course, we do not know which formulae are satisfiable and which have the lowest runtime. With an infinite number of processors we could find in the smallest possible time a satisfying assignment for one of the formulae: let each processor $i \in \{0, 1, 2, \ldots\}$ test the satisfiability of the formula for i time points. However, we do not have an infinite number of processors, and we cannot even simulate an infinite number of processors running at the same speed by a finite number of processors. But we can approximate this scheme.

Our first algorithm uses a finite number n of processes/processors. Our second algorithm uses one process/processor to simulate an infinite number of processes but the simulation

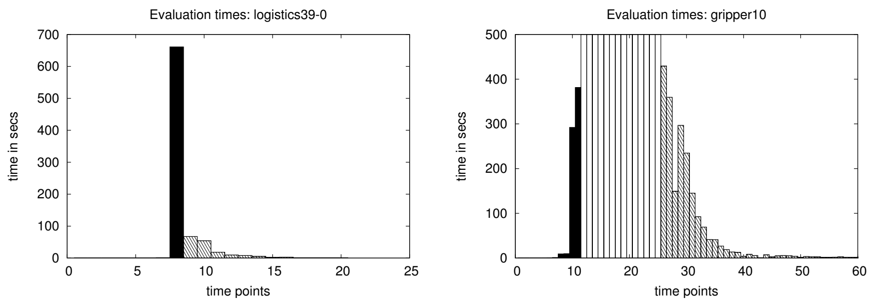


Fig. 6. SAT solver runtimes for two problem instances and different plan lengths

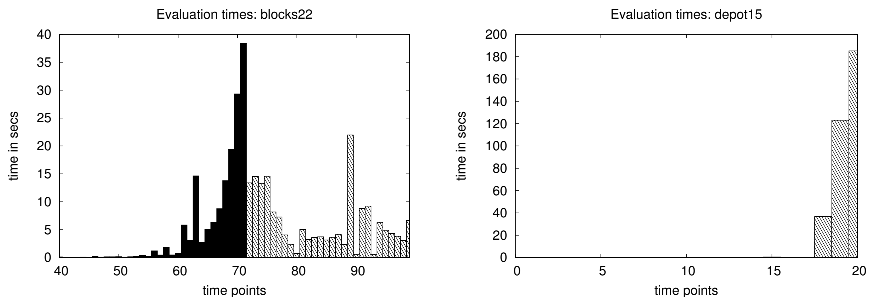


Fig. 7. SAT solver runtimes for two problem instances and different plan lengths

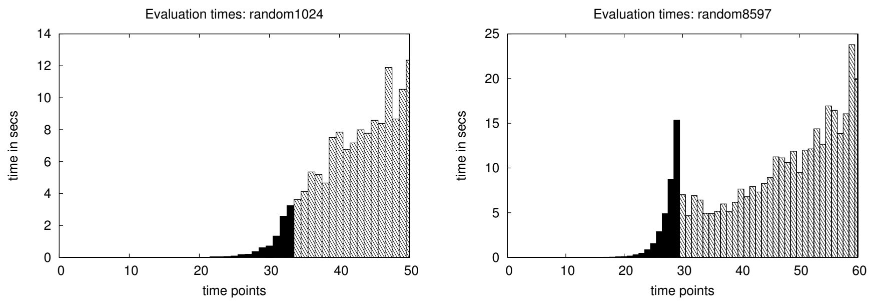


Fig. 8. SAT solver runtimes for two problem instances and different plan lengths

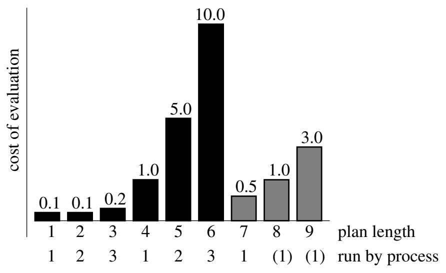


Fig. 9. Evaluation cost of the unsatisfiable formulae for plan lengths 1 to 6 and the satisfiable formulae for plan length 7 and higher. With 3 processes, process 1 finds the first plan (satisfying assignment) after evaluating the formulae for plan lengths 1, 4 and 7 in 0.1+1+0.5 = 1.6 seconds. This is $3 \times 1.6 = 4.8$ seconds of total CPU time. The sequential strategy needs $0.1 + 0.1 + 0.2 + 1 + 5 + 10 + 0.5 = 16.9$ seconds. With 4 processes a plan would be found by process 3 in $0.2 + 0.5 = 0.7$ seconds, which is $4 \times 0.7 = 2.8$ seconds of total CPU time.

runs the processes at variable rates so that for every formula ϕ_t and every $k \geq 0$ there is a time point when k seconds of CPU time has been spent for testing the satisfiability of ϕ_t. If all processes were run at the same rate, this property could not be fulfilled.

Except for the rightmost diagram in Figure 7 and the leftmost diagram in Figure 8, the diagrams depict steeply growing costs of determining unsatisfiability of a sequence of formulae followed by small costs of determining satisfiability of formulae corresponding to plans. This pattern could be abstracted as the diagram in Figure 9. The strategy implemented by our first algorithm distributes the computation to n concurrent processes and initially assigns the first n formulae to the n processes. Whenever a process finds its formula satisfiable, the computation is terminated. Whenever a process finds its formula unsatisfiable, the process is given the first unevaluated formula to evaluate. This strategy can avoid completing the evaluation of many of the expensive unsatisfiable formulae, thereby saving a lot of computation effort.

An inherent property of the problem is that unsatisfiable (respectively satisfiable) formulae later in the sequence are in general more expensive to evaluate than earlier unsatisfiable (respectively satisfiable) formulae. The difficulty of the unsatisfiable formulae increases as i increases because the formulae become less constrained, contradictions are not found as quickly, and search trees grow exponentially. The increase in the difficulty of satisfiable formulae is less clear. For example, for the first satisfiable formula ϕ_s there may be few plans while for later formulae there may be many plans, and the formulae would be less constrained and easier to evaluate. However, as formula sizes increase, the possibility of getting lost in parts of the search space that do not contain any solutions also increases. Therefore an increase in plan length also later leads to an increase in difficulty.

The new algorithms are useful if a peak of difficult formulae precedes easier satisfiable formulae, for example when it is easier to find a plan of length n than to prove that no plans of length $n-1$ exists, or if the first strongly constrained satisfiable formulae corresponding to the shortest plans are more difficult to evaluate than some of the later less constrained ones. The experiments in Section 5.5 show that for many problems one or both of these conditions hold.

We discuss the standard sequential algorithm and the two new algorithms in detail next.

```
procedure AlgorithmS()
$i$ := 0;
repeat
    test satisfiability of $\phi_i$;
    if $\phi_i$ is satisfiable then terminate;
    $i$ := $i + 1$;
until 1=0;
```

Fig. 10. Algorithm S

```
procedure AlgorithmA($n$)
$P$ := $\{\phi_0, \dots, \phi_{n-1}\}$;
uneval := $n$;
repeat
    $P'$ := $P$;
    for each $\phi \in P'$ do
        continue evaluation of $\phi$ for $\epsilon$ seconds;
        if $\phi$ was found satisfiable then goto finish;
        if $\phi$ was found unsatisfiable then
            $P$ := $P \cup \{\phi_{\text{uneval}}\} \backslash \{\phi\}$;
            uneval := uneval + 1;
        end if
    end do
until 0=1
finish:
```

Fig. 11. Algorithm A

5.1 Algorithm S: sequential evaluation

The standard algorithm for finding plans in the satisfiability and related approaches to planning tests the satisfiability of formulae for plan lengths 0, 1, 2, and so on, until a satisfiable formula is found [Blum and Furst 1997; Kautz and Selman 1996]. This algorithm is given in Figure 10. This algorithm, like the ones discussed next, can be extended so that it terminates whenever no plans exist. This is by the observation that with n Boolean state variables there are at most 2^n reachable states and hence if a plan exists, then a plan of length less than 2^n exists. This, however, provides only an impractical termination test. More practical tests exist [Sheeran, Singh, and Stålmarck 2000; McMillan 2003; Mneimneh and Sakallah 2003].

5.2 Algorithm A: multiple processes

The first new algorithm (Figure 11) which we call Algorithm A is based on parallel or interleaved evaluation of a fixed number n of formulae by n processes. As the special case $n = 1$ we have Algorithm S. Whenever a process finishes the evaluation of a formula, it is given the first unevaluated formula to evaluate. The constant ϵ determines the coarseness of CPU time division during the evaluation. The *for each* loop in this algorithm and in the next algorithm can be implemented so that several processors are used in parallel.

There is a simple improvement to the algorithm: when formula ϕ_i is found unsatisfiable, the algorithm terminates the evaluation of all ϕ_j for $j < i$ because they must all be unsatisfiable. However, this modification does not usually have any effect because of the monotonically increasing evaluation cost of the unsatisfiable formulae: ϕ_j would already have been found unsatisfiable when ϕ_i with $i > j$ is found unsatisfiable. We ignore this improvement in the following.

```
procedure AlgorithmB($\gamma$)
t := 0;
for each $i \geq 0$ do done[$i$] = false;
for each $i \geq 0$ do time[$i$] = 0;
repeat
    $t := t + \delta$;
    for each $i \geq 0$ such that done[$i$] = false do
        if time[$i$] $+ n\epsilon \leq t\gamma^i$ for some maximal $n \geq 1$ then
            continue evaluation of $\phi_i$ for $n\epsilon$ seconds;
            if $\phi_i$ was found satisfiable then goto finish;
            time[$i$] := time[$i$] $+ n\epsilon$;
            if $\phi_i$ was found unsatisfiable then done[$i$] := true; end if
        end if
    end do
until 0=1
finish:
```

Fig. 12. Algorithm B

5.3 Algorithm B: geometric division of CPU use

In Algorithm A the choice of n is determined by the (assumed) width and height of the peak preceding the first satisfiable formulae, and our experiments indicate that small differences in n may make a substantial difference in the runtimes: consider for example the problem instance logistics39-0 in Figure 6 for which runtime of Algorithm A with $n = 1$ is more than 10 times the runtime with $n = 2$.

Our second algorithm which we call Algorithm B addresses the difficulty of choosing the value n in Algorithm A. Algorithm B evaluates in an interleaved manner an unbounded number of formulae. The amount of CPU given to each formula depends on its index: if formula ϕ_k is given t seconds of CPU during a certain time interval, then a formula $\phi_i, i \geq k$ is given $\gamma^{i-k}t$ seconds. This means that every formula gets only slightly less CPU than its predecessor, and the choice of the exact value of the constant $\gamma \in]0, 1[$ is far less critical than the choice of n for Algorithm A.

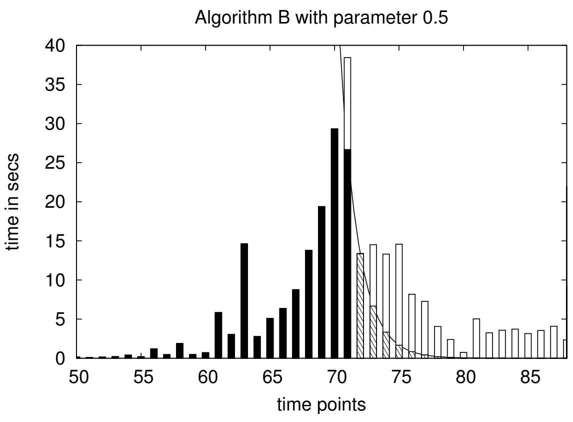
Algorithm B is given in Figure 12. Variable t, which is repeatedly increased by δ, characterizes the total CPU time $\frac{t}{1-\gamma}$ available so far. As the evaluation of ϕ_i proceeds only if it has been evaluated for at most $t\gamma^i - \epsilon$ seconds, CPU is actually consumed less than $\frac{t}{1-\gamma}$, and there will be at time $\frac{t}{1-\gamma}$ only a finite number $j \leq \log_\gamma \frac{\epsilon}{t}$ of formulae for which evaluation has commenced.

In a practical implementation of the algorithm, the rate of increase δ of t is increased as the computation proceeds; otherwise the inner *foreach* loop would later often be executed without evaluating any of the formulae further. We could choose δ for example so that the first unfinished formula ϕ_i is evaluated further at every iteration ($\delta = \frac{\epsilon}{\gamma^i}$).

The constants n and γ respectively for Algorithms A and B are roughly related by $\gamma = 1 - \frac{1}{n}$: of the CPU capacity $\frac{1}{n} = 1 - \gamma$ is spent evaluating the first unfinished formula, and the lower bound for Algorithm B is similarly related to the lower bound for Algorithm A. Algorithm S is the limit of Algorithm B when γ goes to 0.

5.4 Properties of the algorithms

We analyze the properties of the algorithms.



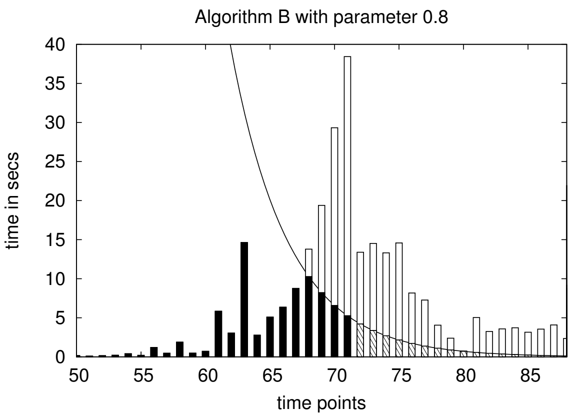


Fig. 13. Illustration of two runs of Algorithm B. When $\gamma = 0.5$ most CPU time is spent evaluating the first formulae, and when the first satisfiable formula is detected also the unsatisfiability of most of the preceding unsatisfiable formulae has been detected. With $\gamma = 0.8$ more CPU is spent for the later easier satisfiable formulae, and the expensive unsatisfiability tests have not been completed before finding the first satisfying assignment.

Definition 5.1 (Speed-up) *The* speed-up *of an algorithm X (with respect to Algorithm S) is the ratio of the runtimes of Algorithm S and the Algorithm X.*

If the speed-up is greater than 1, then the algorithm is faster than Algorithm S.

In our analysis we assume that the constant ϵ in Algorithm A is infinitesimally small, and hence, after a process finishes with one formula, the evaluation of the next formula starts immediately, and the algorithm terminates immediately after a satisfiable formula is found.

If there is no peak because the last unsatisfiable formulae are not more difficult than some of the first satisfiable ones, then Algorithm A with $n \geq 2$ may need n times more CPU than Algorithm S because $n-1$ satisfiable formulae are evaluated unnecessarily. We formally establish worst-case bounds for Algorithm A.

Theorem 5.2 *The speed-up of Algorithm A with n processes is at least $\frac{1}{n}$. This lower bound is strict.*

PROOF. The worst case $\frac{1}{n}$ can show up in the following situation. Assume the first satisfiable formula is evaluated in time t, the preceding unsatisfiable formulae are evaluated in time 0, and the following satisfiable formulae are evaluated in time $\geq t$. Then the total runtime of Algorithm A is tn, while the total runtime of Algorithm S is t.

Assume the runtimes (CPU time) for the formulae are $t_0, t_1, \ldots, t_s, \ldots$, and ϕ_s is the first satisfiable formula. The total runtime of Algorithm S is $\sum_{i=0}^{s} t_i$. This is also an upper bound on the CPU time consumed by Algorithm A on $\phi_0, \ldots, \phi_s$. Additionally, Algorithm A may spend CPU evaluating $\phi_{s+1}, \phi_{s+2}, \ldots$. The evaluation of these formulae starts at the same time or later than the evaluation of the first satisfiable formula ϕ_s. As $n-1$ processes may spend all their time evaluating these formulae after the evaluation of ϕ_s has started, the total CPU time spent evaluating them may be at most $(n-1)t_s$. Hence Algorithm A spends CPU time at most

$$\sum_{i=0}^{s} t_i + (n-1)t_s$$

in comparison to

$$\sum_{i=0}^{s} t_i$$

with Algorithm S. The speed-up is therefore at least

$$\frac{\sum_{i=0}^{s} t_i}{\sum_{i=0}^{s} t_i + (n-1)t_s} = \frac{1}{1+(n-1)\frac{t_s}{\sum_{i=0}^{s} t_i}} \geq \frac{1}{1+n-1} = \frac{1}{n}.$$

□

In the other direction, there is no finite upper bound on the speed-up of Algorithm A in comparison to Algorithm S for any number of processes $n \geq 2$. Consider a problem instance with evaluation time t_0, t_1 and t_2 respectively for the first three formulae, the first two of which are unsatisfiable and the third satisfiable. Let $t_0 = t_2$ and $t_1 = ct_2$. The constant c could be arbitrarily high. Algorithm S runs in $(c+2)t_2$ time, while Algorithm A with $n = 2$ runs in $2t_2$ time. Hence the speed-up $\frac{c+2}{2}$ can be arbitrarily high.

Next we analyze the properties of Algorithm B assuming that the constants δ and ϵ are infinitesimally small and the evaluation of all of the formulae ϕ_i therefore proceeds continuously at rate γ^i.

Theorem 5.3 *The speed-up of Algorithm B is at least* $1-\gamma$. *This lower bound is strict.*

PROOF. As with Algorithm A the worst case is reached when all unsatisfiable formulae preceding the first satisfiable formula ϕ_s are evaluated and the evaluation of many of the satisfiable ones has proceeded far. The disadvantage in comparison to Algorithm S is the unnecessary evaluation of many of the satisfiable formulae. Hence Algorithm B spends CPU time at most

$$\sum_{i=0}^{s} t_i + \sum_{i\geq 1} t_s\gamma^i = \sum_{i=0}^{s} t_i + \frac{1}{1-\gamma}t_s - t_s$$

in comparison to

$$\sum_{i=0}^{s} t_i$$

with Algorithm S. The speed-up is therefore at least

$$\begin{aligned}\frac{\sum_{i=0}^{s} t_i}{\sum_{i=0}^{s} t_i + \frac{1}{1-\gamma}t_s - t_s} &= \frac{1}{1+\frac{\frac{1}{1-\gamma}t_s - t_s}{\sum_{i=0}^{s} t_i}} \geq \frac{1}{1+\frac{\frac{1}{1-\gamma}t_s - t_s}{t_s}} \\ &= \frac{1}{1+\frac{1}{1-\gamma}-1} = 1-\gamma.\end{aligned}$$

This lower bound is strict: if ϕ_i is satisfiable, evaluation times for $\phi_j, j < i$ are 0, and evaluation times for $\phi_i, i > 1$ are not lower than that of ϕ_1, then the speed-up is only $1-\gamma$. $\square$

The worst-case speed-ups of these algorithms are the same if we observe the equation $\gamma = 1 - \frac{1}{n}$ relating their parameters.

Algorithm B does not have plan quality guarantees but Algorithm A has.

Theorem 5.4 *If a plan exists, Algorithm A with parameter* $n \geq 1$ *is guaranteed to find a plan that is at most* $n-1$ *steps longer than the shortest existing one.*

PROOF. So assume Algorithm A finds a plan with t steps. This means that the process for formula ϕ_t determined that the formula is satisfiable. There are at most $n-1$ processes for formulae ϕ_s with $s < t$, and all formulae ϕ_s for $s < t$ for which a process terminated are unsatisfiable.

All formulae preceding an unsatisfiable formula are unsatisfiable. Consider formula ϕ_{t-n}.

If the process evaluating ϕ_{t-n} has terminated, the formula must have been unsatisfiable, and hence the plan from ϕ_t is at most $n-1$ steps longer than the shortest existing one which much have length over $t-n$.

If the process evaluating ϕ_{t-n} has not terminated, then the evaluation of one of the $n-1$ formulae $\phi_{t-n+1}, \ldots, \phi_{t-1}$ must already have been terminated, because there are n processes and two of them were evaluating ϕ_{t-n} and ϕ_t. Since ϕ_t was the first one found satisfiable, one of the formulae $\phi_{t-n+1}, \ldots, \phi_{t-1}$ that was evaluated was unsatisfiable, and hence the formula ϕ_{t-n} must also be unsatisfiable, yielding the same lower bound for the plan length. $\square$

5.5 Experiments

Algorithms A and B increase efficiency for problem instances sampled from the set of all problem instances (Section 4.3). The improvements in comparison to Algorithm S are biggest for easy problem instances right of the hardest part of the phase transition region with 100 state variables or more. For the most difficult instances in the middle of the phase transition region the satisfiable formulae are often as difficult as the unsatisfiable ones and hence Algorithms A and B do not seem to bring as much benefit. We did not carry out exhaustive experimentation because of the extremely high computational resource consumption and the difficulty to derive exact characterizations of the performance improvement when most of the problem instances could not be practically solved by using Algorithm S.

We illustrate properties of the algorithms on a collection of problems from the AIPS planning competitions. Plans for most of these problems can be found in polynomial time by simple domain-specific algorithms, and planners using heuristic search [Bonet and Geffner 2001] have excelled on these problems, while they had been considered difficult for planners based on satisfiability testing or CSP techniques. In this section we demonstrate that the new algorithms change this situation.

Many of these benchmark problems follow the same scheme in which objects are transported with vehicles from their initial locations to their target locations (Logistics, Depots, DriverLog, ZenoTravel, Gripper, Elevator), with one of them (Depots) combining transportation with stacking objects as in the well-known Blocks World problem. Some others (Satellite, Rovers) are variations of the transportation scenario in which different locations are visited to carry out some tasks. Some of the benchmark problems have the form of a scheduling problem (Elevator, Schedule) but do not involve any restrictions on resource consumption and therefore only test the property of feasibility which for these problems can be tested in low polynomial time by a simple algorithm.

To demonstrate the usefulness of the algorithms for a wider range of problems, in addition to the planning competition problems which are solvable by simple domain-specific algorithms in polynomial time, we also consider hard instances of an NP-hard planning problem. The planning competition problems are easy because they do not make restrictions on resource consumption and satisfying one subgoal never makes it more difficult to satisfy another. Hence we also consider a planning problem with critical restrictions on resource consumption. We call this problem the Mechanic problem. The objective is to perform a maintenance operation to a fleet of n aircraft. The aircraft fly according to some schedule and visit one of three airports five times during a time period of t days. A mechanic/equipment can be present at one of the three airports on any given day, and can perform the maintenance operation to all aircraft visiting that airport that day. This problem can be viewed as a form of a set covering problem but we make it a sequential decision making problem so that we can talk about completing the maintenance within $t' \leq t$ first days of the time period. We solve the problem for a $t = 30, 40, 50, ...$ and set $n = 3t$. With $n = 3t$ the problem is rather strongly constrained but still usually soluble. For low n it is easier to find a plan because there are only few aircraft, and for much higher n there are too many aircraft and no plan necessarily exists.

For each problem instance we generate formulae for plan lengths up to 10 or 30 beyond the first (assumed) satisfiable formula according to the $\exists$-step semantics encoding in Section 3.4.4. We use the linear-size encoding of the parallelism constraints if it is less than half of the size of the quadratic encoding that does not require introducing auxiliary propositional variables to avoid exceeding Siege's upper bound of 524288 propositional variables.

Then we test the satisfiability of every formula and cancel the run if the satisfiability is not determined in 60 minutes of CPU time. Like in the experiments in Section 4, we use the Siege V4 SAT solver by Lawrence Ryan of the Simon Fraser University on a 3.6 GHz Intel Xeon computer.

Then we compute from the runtimes of all these formulae the total runtimes under algorithms A and B with different values for the parameters n and γ. Algorithm S is the special case $n = 1$ of Algorithm A. The constants ϵ and δ determining the granularity of CPU time division are set infinitesimally small. Formulae that are beyond the plan-length horizon or that take over 60 minutes to evaluate are considered as having infinite evaluation time. The times do not include generation of the formulae. The two expensive parts of the formula generation are the computation of the invariants and the disabling graph. For most of the benchmark problem instances these both take a fraction of a second, but for some of the biggest instances of the Logistics, Depot, and Driver problems 10 or 20 seconds, and a total of 6 minutes for the biggest ZenoTravel instance and 3 minutes for the biggest Logistics instance. A more efficient implementation would bring these times down to seconds.

The runtimes for a number of problems from the AIPS planning competitions of 1998, 2000 and 2002 and for the Mechanic problem are given in Table VIII. For most benchmarks we give the runtimes of the most difficult problem instances, which in some cases are the last ones in the series, as well as some of the easier ones. Most of the runtimes that are not given in the table are below one second for every evaluation strategy. Some of the benchmark series cannot be solved until the end efficiently, and we give data just for some of the most difficult instances that can be solved. We discuss these benchmark problems below.

The column "easiest" gives the shortest time it took to determine the satisfiability of any of the satisfiable formulae corresponding to a plan. These times in almost all cases are very low, even when the total runtime of Algorithms S, A and B is high. Hence in almost all cases the total runtime is strongly dominated by the unsatisfiability tests.

	Algorithm A with n					Algorithm B with γ				easiest	HSP	FF
instance	1	2	4	8	16	0.5000	0.7500	0.8750	0.9375			
block-18-0	8.6	7.8	7.6	**5.8**	6.6	8.0	7.9	**7.7**	9.3	0.1	6.4	–
block-20-0	**11.3**	12.2	13.8	16.9	15.5	13.0	16.5	20.1	18.3	0.2	82.1	0.0
block-22-0	122.4	106.9	96.7	77.0	**35.4**	106.0	62.2	33.5	**27.0**	0.3	79.6	0.5
block-24-0	2877.5	2675.7	1854.0	829.0	**167.4**	2087.3	583.3	284.8	**246.8**	0.7	–	–
block-26-0	5347.5	5000.0	4640.1	3103.9	**539.0**	4116.7	1140.0	242.6	**126.3**	0.9	46.8	0.0
block-28-0	3447.8	3413.4	3246.8	1984.3	**813.3**	2867.0	1746.1	1027.6	**336.4**	1.1	–	27.2
block-30-0	–	–	13949.9	7541.0	**6349.1**	13934.0	6577.4	1717.4	**503.9**	1.9	–	0.0
block-32-0	–	–	–	28695.4	**14326.9**	$> 27h$	36417.3	8182.8	**2245.7**	11.3	–	0.0
block-34-0	227.6	227.8	224.2	231.5	**208.8**	238.4	248.2	264.6	**188.5**	1.9	–	0.1
driver-4-4-8	**0.5**	0.6	**0.3**	0.5	0.8	0.6	0.6	0.7	1.1	0.1	2.9	0.1
driver-5-5-10	731.2	549.5	631.6	**237.7**	440.2	969.8	507.0	**472.4**	651.1	27.5	–	–
driver-5-5-15	72.4	**36.1**	50.4	100.4	200.6	**56.0**	72.7	120.5	219.8	12.5	72.21	–
driver-5-5-20	1018.2	690.1	792.4	940.7	**17.8**	967.5	148.2	35.4	**24.0**	0.5	1428.0	–
driver-5-5-25	–	6433.9	**2218.9**	3542.3	4132.2	4553.4	**4100.7**	5800.5	7865.5	258.2	–	609.5
driver-8-6-25	–	–	13333.9	**11081.4**	22162.6	27447.3	24120.5	**22377.1**	31375.3	1385.2	859.0	–
satel-12	31.1	5.1	**1.4**	1.8	2.7	4.0	**2.5**	3.1	4.6	0.2	3.3	0.1
satel-13	**14.8**	**14.2**	18.2	14.9	17.9	21.0	29.0	24.1	22.8	0.5	10.1	0.3
satel-19	45.1	28.4	21.6	**5.0**	5.6	42.3	13.1	**9.4**	10.1	0.3	8.8	0.3
satel-20	–	1806.4	266.6	**33.0**	35.0	187.1	69.3	**55.3**	63.5	2.1	23.2	4.9
gripper-5	3443.2	1053.7	35.5	7.2	**5.0**	31.7	16.2	2.1	**0.9**	0.0	0.0	0.0
gripper-6	–	–	2679.6	23.4	**10.4**	121.9	45.6	4.1	**1.7**	0.0	0.0	0.0
gripper-7	–	–	–	491.3	**28.3**	1968.0	128.2	7.9	**2.8**	0.0	0.0	0.0
gripper-8	–	–	–	13285.5	**293.1**	57298.9	790.1	27.3	**4.7**	0.0	0.0	0.0
gripper-9	–	–	–	–	**832.6**	$> 27h$	589.7	37.7	**13.0**	0.1	0.1	0.0
gripper-10	–	–	–	–	**216.3**	31496.5	569.3	126.8	**17.1**	0.1	0.1	0.0
gripper-11	–	–	–	–	–	$> 27h$	87479.2	2308.0	**335.4**	0.8	0.1	0.0
gripper-12	–	–	–	–	–	$> 27h$	$> 27h$	8306.4	**1117.5**	0.8	0.1	0.0
zeno-5-10	0.3	0.3	**0.2**	0.2	0.5	0.3	**0.2**	0.3	0.6	0.0	24.2	0.1
zeno-5-15	154.2	77.1	8.7	**2.3**	4.5	17.7	5.1	**4.8**	6.6	0.3	18.5	0.3
zeno-5-15b	40.5	25.3	**7.1**	9.4	9.1	24.4	**14.6**	17.6	17.7	0.5	104.5	0.4
zeno-5-20	–	–	9036.9	6422.6	**2896.0**	16459.9	1364.2	126.8	**64.8**	1.1	188.4	1.4
zeno-5-20b	–	–	–	**10822.9**	18744.6	87164.6	23385.8	**21683.0**	30471.3	1171.5	411.5	1.4
zeno-5-25	–	–	–	**12987.1**	25914.9	$> 27h$	37341.0	**29810.9**	39109.3	1619.7	332.4	4.4
sched-33-0	79.0	53.7	13.0	**5.0**	6.7	22.8	10.9	**10.1**	11.3	0.2	–	0.7
sched-35-0	2225.2	1435.5	19.5	3.6	**2.9**	14.3	7.8	**4.9**	5.2	0.2	–	0.7
sched-37-0	346.2	184.4	92.8	**8.6**	9.6	80.4	24.2	**19.4**	19.5	0.6	–	0.5
sched-39-0	–	–	–	592.2	**140.3**	5889.8	1084.6	437.6	**221.9**	1.9	–	1.7
sched-41-0	–	–	–	479.1	**35.4**	3040.7	237.1	91.7	**80.7**	1.3	–	1.0
sched-43-0	–	1565.2	23.9	**11.6**	17.4	47.3	**20.0**	21.4	23.7	0.4	–	2.2
sched-45-0	–	–	1398.1	109.5	**41.6**	786.6	257.8	100.2	**73.3**	1.5	–	1.0
sched-47-0	–	–	–	14066.9	**245.0**	62768.3	1708.6	607.0	**215.4**	2.2	–	4.3
sched-49-0	–	–	–	9511.7	**561.6**	24913.2	2609.9	426.4	**169.2**	2.1	–	6.0
sched-51-0	–	–	–	–	**1151.2**	$> 27h$	8327.0	1692.6	**889.2**	7.6	–	3.1
depot-09-5451	**14.1**	24.8	43.9	85.8	171.5	24.8	46.3	89.1	174.8	10.7	–	0.7
depot-12-9876	**255.4**	509.7	1018.6	2036.9	4073.6	509.9	1019.1	2037.5	4074.2	254.6	–	3.1
depot-15-4534	**42.8**	79.3	154.8	305.4	609.9	80.9	157.1	309.6	614.4	38.1	–	3.4
depot-18-1916	**5.9**	11.2	21.9	43.5	86.9	11.4	22.2	43.9	87.4	5.4	–	0.8
depot-19-6178	**0.2**	0.2	0.3	0.4	0.6	0.3	0.3	0.5	0.8	0.0	–	0.2
depot-20-7615	34.2	66.7	131.9	262.1	**10.4**	67.0	35.6	**18.5**	18.7	0.4	–	14.3
depot-21-8715	0.2	**0.1**	0.2	0.3	0.6	**0.2**	0.3	0.4	0.7	0.0	51.4	0.3
depot-22-1817	**27.1**	50.8	98.9	194.8	389.4	51.4	100.1	197.5	392.2	24.3	–	55.3
log-20-0	3.4	2.8	0.6	0.7	**0.5**	1.3	1.1	0.9	**0.8**	0.0	2.2	0.1
log-24-0	0.9	**0.4**	0.6	1.0	1.6	**0.6**	0.8	1.2	1.8	0.0	3.1	0.1
log-28-0	87.7	53.3	13.8	**1.9**	3.5	15.0	4.1	**3.8**	3.9	0.1	28.0	1.2
log-32-0	–	53.1	18.9	37.4	**16.3**	37.9	33.7	26.6	**16.6**	0.3	43.4	4.5
log-36-0	–	101.1	20.2	30.1	**11.8**	58.7	46.5	29.2	**14.5**	0.2	81.1	2.6
log-40-0	–	–	111.2	**4.6**	7.2	37.5	10.6	**9.9**	13.5	0.4	267.8	4.5
log-41-0	–	–	52.4	20.0	**5.4**	175.3	14.8	**9.1**	9.8	0.3	247.1	4.2
mechanic-30-90	0.5	0.4	**0.3**	0.4	0.3	**0.4**	0.4	0.4	0.5	0.0	4.1	0.1
mechanic-40-120	0.5	**0.4**	0.5	0.6	0.5	**0.5**	0.6	0.6	0.6	0.0	14.1	0.2
mechanic-50-150	1.1	**1.0**	1.0	1.7	2.9	**1.2**	1.4	2.0	2.7	0.1	226.7	0.4
mechanic-60-180	13.1	6.9	3.3	2.0	**1.0**	4.3	1.6	**1.2**	1.3	0.0	56.46	0.7
mechanic-70-210	4.5	2.9	2.4	**1.0**	1.3	3.3	1.8	**1.5**	1.8	0.1	–	–
mechanic-80-240	2.0	3.0	**1.1**	1.5	2.4	2.4	**1.7**	2.0	2.8	0.1	213.9	3.1
mechanic-90-270	15.1	**2.1**	2.4	3.2	2.9	**3.0**	3.1	3.9	3.9	0.1	339.0	3.6
mechanic-100-300	77.0	47.2	52.2	17.3	**8.0**	64.5	37.5	25.6	**14.7**	0.2	–	–
mechanic-110-330	162.0	192.5	117.8	81.4	**42.2**	164.7	90.2	64.6	**40.3**	0.6	–	–
mechanic-120-360	991.4	717.5	273.5	61.3	**30.6**	185.8	72.8	**56.6**	62.4	0.9	–	–

Table VIII. Runtimes of Algorithms A and B. Column $n = 1$ is Algorithm S. Dash indicates a missing upper bound on the runtime when some formulae were not evaluated in 60 minutes. The best runtimes for Algorithms A and B are highlighted for each problem instance (sometimes this is the special case Algorithm S.) The column “easiest” shows the lowest runtime for any of the satisfiable formulae.

	Algorithm A with n					Algorithm B with ϵ			
instance	1	2	4	8	16	0.500	0.750	0.875	0.938
blocks-34-0	124 / 124	125 / 124	125 / 124	125 / 124	125 / 124	125 / 124	125 / 124	135 / 129	135 / 129
driver-8-6-25	–	–	12 / 193	14 / 206	14 / 206	11 / 178	14 / 206	14 / 206	14 / 206
satell-20	–	10 / 230	11 / 166	12 / 285	17 / 321	11 / 166	12 / 285	15 / 309	15 / 309
gripper-10	–	–	–	–	25 / 65	25 / 65	25 / 65	49 / 150	49 / 150
zeno-5-15b	5 / 87	5 / 87	7 / 98	9 / 140	14 / 191	7 / 98	7 / 98	14 / 191	14 / 191
sched-37-0	13 / 60	13 / 60	13 / 60	16 / 57	20 / 69	14 / 52	16 / 57	16 / 57	20 / 69
depot-19-6178	10 / 98	10 / 98	10 / 98	10 / 98	10 / 98	10 / 98	10 / 98	10 / 98	10 / 98
depot-20-7615	14 / 153	14 / 153	14 / 153	14 / 153	23 / 170	14 / 153	23 / 170	23 / 170	29 / 193
log-20-0	9 / 176	10 / 151	11 / 199	12 / 163	20 / 249	11 / 199	12 / 163	20 / 249	20 / 249
log-28-0	9 / 243	10 / 298	11 / 288	13 / 309	15 / 340	13 / 309	13 / 309	13 / 309	24 / 443

Table IX. Numbers of time points and operators in plans found by Algorithms A and B. Column $n = 1$ is Algorithm S. Dash indicates missing data when some formulae were not evaluated in 60 minutes.

Table IX shows the numbers of time points and operators in the plans obtained for some of the benchmark problems reported in Table VIII. In many cases the easiest satisfiable formulae are not the first ones, and these formulae typically have satisfying assignments that correspond to plans having many useless operators, which for algorithms A and B can lead to plans with many more operators than for algorithm S. However, the benchmarks have a simple structure and these plans with more operators are usually not genuinely different: the additional operators are either irrelevant for reaching the goals or contain pairs of operators and their inverses. It would be easy to eliminate these types of useless operators by a simple postprocessing step.

The Movie, MPrime and Mystery benchmarks from the 1998 competition and Rovers from 2002 are very easy for every evaluation strategy (fraction of a second in most cases) but we cannot produce the biggest MPrime instance because of a memory restriction.

The Logistics (1998 and 2000) and Satellite (2002) series are solved completely. Proving inexistence of plans slightly shorter than the optimal plan length is in some cases difficult but the new evaluation algorithms handle this efficiently.

The Depots (2002) problems are also relatively easy but in contrast to the rest of the benchmarks the new evaluation algorithms in some cases increase the runtimes up to the theoretical worst case.

The DriverLog and ZenoTravel (2002) problems are solved quickly except for some of the biggest instances. We cannot find satisfiable formulae for the last ZenoTravel problem within our time limit[6], and finding plans for the preceding two instances of ZenoTravel and the last two of DriverLog is also slow. The difficulty lies in finding tight lower bounds for plan lengths by determining the unsatisfiability of formulae.

Blocks World (2000) problems lead to very big formulae (size over 100 MB and over 524288 propositions), and we can solve only two thirds of the series.

Elevator (2000), Schedule (2000) and Gripper (1998) are a challenge because only very loose lower bounds on plan length are easy to prove. Finding plans corresponding to a given satisfiable formula is very easy (some seconds at most) but locating these formulae is very expensive. Increasing parameters n and γ improves runtimes.

The formulae generated for FreeCell (2002) are too big (hundreds of megabytes) for the current SAT solvers to solve them efficiently. This benchmark series along with the blocks world problems are the only ones that are not solved almost entirely.

[6]The number of propositions in formulae for plan lengths much higher than the presumed shortest plan length exceeds Siege's upper bound 524288.

All in all, it seems that a conservative use of the new algorithms (especially Algorithm B with $\gamma \in [0.7..0.9]$) leads to a general improvement in the runtimes in comparison to Algorithm S.

Decrease in plan quality is indirectly related to decrease in runtime. This depends on whether the first satisfiable formulae are the easiest ones. In general, satisfying valuations that are found for plan lengths much higher than the shortest plan length correspond to plans with more operators, but not always.

6. RELATED WORK

6.1 Encodings of planning in the propositional logic

Kautz and Selman [1992] introduced the idea of doing planning by using satisfiability algorithms. Following the introduction of the GraphPlan algorithm that successfully utilized parallel plans [Blum and Furst 1997], Kautz and Selman [1996, 1999] extended their approach to parallel plans.

Ernst et al. [1997] investigated different encodings planning in the propositional logic. A difference to other works on planning as satisfiability is that some of the encodings utilize the regularities that are obvious in the schematic representations of operators.

Following the work by Kautz and Selman, translations of planning into other formalisms have been proposed [Dimopoulos, Nebel, and Koehler 1997; Kautz and Walser 1999; Wolfman and Weld 1999; van Beek and Chen 1999; Do and Kambhampati 2001] but all these works – with the exception of Dimopoulos et al. – use the notions of parallel and sequential plans already used by Kautz and Selman.

Dimopoulos et al. [1997] noticed that the notion of parallel plans used by Blum and Furst [1997] can be relaxed to what we have formalized as $\exists$-step semantics. They called this idea *post-serializability* and showed how to transform operators for some planning problems to make them post-serializable. They did not propose a general translation from arbitrary planning problems as we have done in this work. Rintanen [1998] implemented this idea in a constraint-based planner and Cayrol et al. [2001] in the GraphPlan framework.

The preconditions-effects graphs of Dimopoulos et al. [1997] are a subclass of our disabling graphs. Dimopoulos et al. used these graphs for defining a notion of plans similar to our $\exists$-step plans but did not use them for deriving efficient encodings of planning problems. The definition of preconditions-effects graphs often requires many more edges than the definition of disabling graphs does, and consequently the SCCs of the former may be much bigger than the SCCs of the latter. The small size of the SCCs of disabling graphs is often critical for obtaining compact and efficient problem encodings.

Outside planning, an idea similar to $\exists$-step semantics has recently been used by Ogata and Tsuchiya [2004] in the context of 1-safe Petri nets. Khomenko et al. [2005b] have recently used an improved encoding for acyclicity tests as required for the $\exists$-step semantics in Section 3.4.

The fact that the sequence of formulae that encode the plan existence problem for different plan lengths has a certain regularity has been utilized in earlier research on bounded model checking [Eén and Sörensson 2003; Heljanko, Junttila, and Latvala 2005].

6.2 Evaluation algorithms

Algorithm B in Section 5 is new, and the idea of Algorithm A has independently been discovered by Nabeshima et al. and briefly described but not formally analyzed in a short published abstract [2002]. The idea behind the algorithms has some resemblance to parallelized Las Vegas algorithms, see for example the work by Luby and Ertel [1994], and randomized restarts in combinatorial search [Gomes, Selman, Crato, and Kautz 2000], but the problems are not directly related. In our case, we have an infinite sequence of problem instances (existence of a plan of length $0, 1, 2, \ldots$) with a certain presumed runtime profile (exponential growth in runtimes of the unsatisfiable formulae preceding the satisfiable formulae), whereas in the other two works the question is about utilizing the properties of the distribution of runtimes of one problem instance with a randomized algorithm. The concurrent use of several SAT solvers for solving a model checking problem has been considered earlier [Zarpas 2004] but was not analyzed as in our work [Rintanen 2004a].

6.3 Heuristic state-space search planners

Heuristic state-space search has become a very popular approach to non-optimal classical planning [Bonet and Geffner 2001], especially in the planning competition community. The last 8 years have seen the development of a collection of techniques to make state-space search planners more efficient for the standard benchmark problems introduced in the AIPS/ICAPS planning competitions. However, most of these techniques have been directly motivated by the benchmarks themselves, and have not been shown to improve efficiency for computationally more challenging problems, for example ones having critical resource constraints like in scheduling problems. The emphasis has been more in making the planners scalable to bigger instances of the types of problems used in the planning competitions.

Since the emphasis in research on planning with SAT has been in the use of efficient general-purpose satisfiability algorithms, a similar development specific to the standard planning benchmarks has been missing. However, some of the techniques introduced for the planning competitions could be used in connection with a SAT-based planner just as well, for example the use of a fast incomplete planner for solving the simplest problems, and only starting the more powerful general-purpose search algorithm if the simpler incomplete planner has failed. Our experiments suggest that using a SAT-based planner and the techniques developed in this paper could be more powerful for the second complete stage than a planner that uses heuristic state-space search.

We compare the planning as satisfiability approach with our improved problem encodings and new plan search algorithms to two well-known heuristic state-space search planners, HSP by Bonet and Geffner [2001] and FF of Hoffmann and Nebel [2001].

HSP is a pure general-purpose planner for solving classical planning problems. It is not guaranteed to find shortest plans. It is based on state-space search with heuristics that estimate the distances between states.

The basic approach in the FF planner is the same as in HSP, but to obtain a better performance it employs a search algorithm and pruning techniques that were obtained by experimentation with the standard benchmark sets [Hoffmann and Nebel 2001]. This approach for improving planner efficiency on the standard benchmarks is similar to many other recent planners proposed by the planning competition community. Many of the techniques used by FF are not completeness-preserving and the planner therefore switches to a more

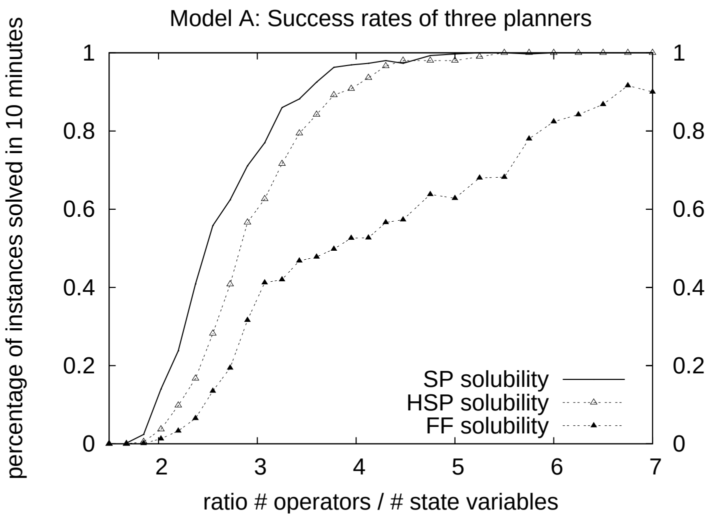


Fig. 14. Percentage of problem instances with 40 state variables solved in 10 minutes by our planner, HSP and FF.

general and complete search algorithm if it is otherwise not able to find a plan. For many of the standard benchmarks the specialized solution techniques are sufficient and lead to extremely good runtimes. However, these techniques cannot be viewed as general-purpose planning techniques and, as we will see below, at least one of them seriously impairs FF's ability to solve problems with a different structure, even for some classes of very easy problem instances.

Figure 14 contains a comparison of our planner (Algorithm S from Section 5.1, encoding from Section 3.4.4), HSP and FF with respect to problem instances from the phase transition region (Section 4.3). For the most difficult problem instances with operators-to-state variables ratio between 2 and 2.5 our planner solves more than twice as many problem instances as HSP or FF with a timeout of 10 minutes. Right of the hardest part of the phase transition region where the problem instances become easier HSP's performance quickly improves. FF, however, has a poor performance even with the very easy instances that have operators-to-state-variables ratio 4 and more: our planner solves all instances in less than a second but FF does not solve up to 40 per cent of these instances in ten minutes. The performance of HSP and FF worsens further relative to our planner when the number of state variables increases [Rintanen 2004b].

The last two columns in Table VIII give runtimes of HSP [Bonet and Geffner 2001] and FF [Hoffmann and Nebel 2001] for some of the standard benchmark problems. These runtimes are not directly comparable to the runtimes of given for planning as satisfiability with the Algorithms S, A and B in the same table because the latter runtimes do not include the time spent in generating the propositional formulae (see Section 5.5 for details.) HSP performs worse than our planner with Blocks World, Schedule, Depot, Logistics and Mechanic. HSP seems to scale better only with Gripper and ZenoTravel. FF's performance is worse or incomparable with Blocks World, Driver and Mechanic. The Blocks World runtimes are better until instance 34 but similarly to our planner FF is not able to solve the last third of the series. Performance of HSP and FF is worse also for some problems that are not listed in the table, for example the Mystery benchmarks from the 1998 competition.

An important factor in the better performance of FF with respect to HSP is the *helpful actions* pruning heuristic that restricts the computation of the heuristic values to only a subset of the successor states of a state [Hoffmann and Nebel 2001]. For this technique to be useful the heuristic estimates have to be very good, which is the case for most of the standard benchmark problems but certainly not for planning problems in general, especially for ones that are inherently difficult. The helpful actions pruning technique is a main factor in the performance difference between FF and HSP for the problem instances in Table VIII because of the high cost of computing the heuristic values for every state. However, it is also responsible for FF's poor performance with the problems in Figure 14: the technique eliminates actions and successor states that are necessary for finding a plan quickly. A version of FF with the pruning technique disabled has a performance much closer to HSP's performance. Hence FF's good performance on many of the standard computationally easy benchmarks has been bought at the price of a dramatically worse performance on structurally more complex problems: the shortcuts FF takes are useful for problems for which the heuristics work well but dramatically fail FF with more difficult problems.

Performance of our planner with Algorithm B and parameter $B = 0.9375$ in comparison to HSP is in most cases as good or better. Performance in comparison to FF is worse on many of the standard benchmark problems but this is due to FF's incomplete techniques, not due to the performance of FF's complete domain-independent search algorithm. FF scales much worse on much smaller but more complex problems, for instance the Mechanic problem.

Results given in Table VIII and in Figure 14 show that planning as satisfiability can indeed be very competitive with heuristic state-space planners. Also, none of the three planners dominate any other, and the strengths of the planners are in different types of problems.

7. CONCLUSIONS

We have given translations of semantics for parallel planning into SAT and shown that one of them, for $\exists$-step semantics, is very efficient, often one or two orders of magnitude faster than previous encodings. This semantics is superior because with our encoding the number of time steps and parallelism constraints is small. Interestingly, the process semantics, a refinement of the standard $\forall$-step semantics that imposes a further condition on plans, typically did not improve planning efficiency in our tests.

The $\exists$-step encoding combined with the novel strategies for finding satisfiable formulae that correspond to plans sometimes leads to substantial improvements in efficiency of planning as satisfiability, and also demonstrate that the approach is for many problems competitive with the fastest planners that are based on heuristic state-space search.

Acknowledgments

We thank Lawrence Ryan for his assistance with Siege.

Keijo Heljanko gratefully acknowledges the financial support of the Academy of Finland (projects 53695, 205431, 206287, 207074, 211025, 213397 and academy research fellow position), FET project ADVANCE contract No IST-1999-29082, and EPSRC grant 93346/01. Ilkka Niemelä gratefully acknowledges the financial support of the Academy of Finland (projects 53695 and 211025).

Jussi Rintanen's research at the Albert-Ludwigs-Universität Freiburg was partly supported by DFG grant RI 1177/2-1 and partly by the National ICT Australia (NICTA) in the framework of the SuperCom project. NICTA is funded through the Australian Government's *Backing Australia's Ability* initiative, in part through the Australian National Research Council.

References

BEST, E. AND DEVILLERS, R. 1987. Sequential and concurrent behavior in Petri net theory. *Theoretical Computer Science 55*, 1, 87–136.

BIERE, A., CIMATTI, A., CLARKE, E. M., AND ZHU, Y. 1999. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems, Proceedings of 5th International Conference, TACAS'99*, W. R. Cleaveland, Ed., *Lecture Notes in Computer Science* vol. 1579 (1999). Springer-Verlag, 193–207.

BLUM, A. L. AND FURST, M. L. 1997. Fast planning through planning graph analysis. *Artificial Intelligence 90*, 1-2, 281–300.

BONET, B. AND GEFFNER, H. 2001. Planning as heuristic search. *Artificial Intelligence 129*, 1-2, 5–33.

BYLANDER, T. 1996. A probabilistic analysis of propositional STRIPS planning. *Artificial Intelligence 81*, 1-2, 241–271.

CAYROL, M., RÉGNIER, P., AND VIDAL, V. 2001. Least commitment in Graphplan. *Artificial Intelligence 130*, 1, 85–118.

CIMATTI, A. 2003. personal communication.

DIEKERT, V. AND MÉTIVIER, Y. 1997. Partial commutation and traces. In *Handbook of Formal Languages*, G. Rozenberg and A. Salomaa, Eds., vol. 3, 457–534. Springer-Verlag.

DIMOPOULOS, Y., NEBEL, B., AND KOEHLER, J. 1997. Encoding planning problems in nonmonotonic logic programs. In *Recent Advances in AI Planning. Fourth European Conference on Planning (ECP'97)*, S. Steel and R. Alami, Eds., Lecture Notes in Computer Science no. 1348 (1997). Springer-Verlag, 169–181.

DO, M. B. AND KAMBHAMPATI, S. 2001. Planning as constraint satisfaction: Solving the planning graph by compiling it into CSP. *Artificial Intelligence 132*, 2, 151–182.

EÉN, N. AND SÖRENSSON, N. 2003. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science 89*, 4, 543–560.

EFRON, B. AND TIBSHIRANI, R. 1986. Bootstrap methods for standard errors, confidence intervals, and other measures of statistical accuracy. *Statistical Science 1*, 54–75.

EFRON, B. AND TIBSHIRANI, R. 1993. *An Introduction to the Bootstrap*. Chapman and Hall, New York.

ERNST, M., MILLSTEIN, T., AND WELD, D. S. 1997. Automatic SAT-compilation of planning problems. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, M. Pollack, Ed. (1997). Morgan Kaufmann Publishers, 1169–1176.

GHALLAB, M., HOWE, A., KNOBLOCK, C., MCDERMOTT, D., RAM, A., VELOSO, M., WELD, D., AND WILKINS, D. 1998. PDDL - the Planning Domain Definition Language, version 1.2. Technical Report CVC TR-98-003/DCS TR-1165 (Oct.), Yale Center for Computational Vision and Control, Yale University.

GOMES, C. P., SELMAN, B., CRATO, N., AND KAUTZ, H. 2000. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning 24*, 1–2, 67–100.

HELJANKO, K. 2001. Bounded reachability checking with process semantics. In *Proceedings of the 12th International Conference on Concurrency Theory (Concur'2001), Lecture Notes in Computer Science* vol. 2154 (2001). Springer-Verlag, 218–232.

HELJANKO, K., JUNTTILA, T. A., AND LATVALA, T. 2005. Incremental and complete bounded model checking for full PLTL. In *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, *Lecture Notes in Computer Science* vol. 3576 (2005). Springer-Verlag, 98–111.

HOFFMANN, J. AND NEBEL, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research 14*, 253–302.

KAUTZ, H. AND SELMAN, B. 1992. Planning as satisfiability. In *Proceedings of the 10th European Conference on Artificial Intelligence*, B. Neumann, Ed. (1992). John Wiley & Sons, 359–363.

KAUTZ, H. AND SELMAN, B. 1996. Pushing the envelope: planning, propositional logic, and stochastic search. In *Proceedings of the 13th National Conference on Artificial Intelligence and the 8th Innovative Applications of Artificial Intelligence Conference* (Aug. 1996). AAAI Press, 1194–1201.

KAUTZ, H. AND SELMAN, B. 1999. Unifying SAT-based and graph-based planning. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, T. Dean, Ed. (1999). Morgan Kaufmann Publishers, 318–325.

KAUTZ, H. AND WALSER, J. 1999. State-space planning by integer optimization. In *Proceedings of the 16th National Conference on Artificial Intelligence (AAAI-99) and the 11th Conference on Innovative Applications of Artificial Intelligence (IAAI-99)* (1999). AAAI Press, 526–533.

KHOMENKO, A., VICTOR AN KONDRATYEV, KOUTNY, M., AND VOGLER, W. 2005a. Merged processes - a new condensed representation of Petri net behaviour. Technical report CS-TR 884 (Jan.), School of Computing Science, University of Newcastle upon Tyne.

KHOMENKO, A., VICTOR AN KONDRATYEV, KOUTNY, M., AND VOGLER, W. 2005b. Merged processes - a new condensed representation of Petri net behaviour. In *CONCUR 2005 - Concurrency Theory, 16th International Conference, CONCUR 2005, San Francisco, CA, USA, August 23-26, 2005, Proceedings*, M. Abadi and L. de Alfaro, Eds., *Lecture Notes in Computer Science* vol. 3653 (2005). Springer-Verlag, 338–352.

LUBY, M. AND ERTEL, R. 1994. Optimal parallelization of Las Vegas algorithms. In *Proceedings of the Annual Symposium on the Theoretical Aspects of Computer Science (STACS'94)* (1994). Springer-Verlag, 463–474.

MCMILLAN, K. L. 2003. Interpolation and SAT-based model checking. In *Proceedings of the 15th International Conference on Computer Aided Verification (CAV 2003)*, W. A. Hunt Jr. and F. Somenzi, Eds., Lecture Notes in Computer Science no. 2725 (2003). 1–13.

MNEIMNEH, M. AND SAKALLAH, K. 2003. Computing vertex eccentricity in exponentially large graphs: QBF formulation and solution. In *SAT 2003 - Theory and Applications of Satisfiability Testing*, E. Giunchiglia and A. Tacchella, Eds., Lecture Notes in Computer Science no. 2919 (2003). 411–425.

NABESHIMA, H., IWANUMA, K., AND INOUE, K. 2002. Effective SAT planning by speculative computation. In *AI 2002: Advances in Artificial Intelligence: 15th Australian Joint Conference on Artificial Intelligence, Canberra, Australia, December 2-6, 2002. Proceedings*, R. I. McKay and J. Slaney, Eds., Lecture Notes in Computer Science no. 2557 (2002). Springer-Verlag, 726–727.

OGATA, S., TSUCHIYA, T., AND KIKUNO, T. 2004. SAT-based verification of safe Petri nets. In *Automated Technology for Verification and Analysis: Second Internation Conference, ATVA 2004, Taipei, Taiwan, ROC, October 31-November 3, 2004. Proceedings*, F. Wang, Ed., Lecture Notes in Computer Science no. 3299 (2004). Springer-Verlag, 79–92.

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

RINTANEN, J. 2004a. Evaluation strategies for planning as satisfiability. In *ECAI 2004: Proceedings of the 16th European Conference on Artificial Intelligence*, R. López de Mántaras and L. Saitta, Eds., *Frontiers in Artificial Intelligence and Applications* vol. 110 (2004). IOS Press, 682–687.

RINTANEN, J. 2004b. Phase transitions in classical planning: an experimental study. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Ninth International Conference (KR 2004)*, D. Dubois, C. A. Welty, and M.-A. Williams, Eds. (2004). AAAI Press, 710–719.

RINTANEN, J. 2005. State-space traversal techniques for planning. Report 220, Albert-Ludwigs-Universität Freiburg, Institut für Informatik.

RINTANEN, J., HELJANKO, K., AND NIEMELÄ, I. 2004. Parallel encodings of classical planning as satisfiability. In *Logics in Artificial Intelligence: 9th European Conference, JELIA 2004, Lisbon, Portugal, September 27-30, 2004. Proceedings*, J. J. Alferes and J. Leite, Eds., Lecture Notes in Computer Science no. 3229 (2004). Springer-Verlag, 307–319.

SHEERAN, M., SINGH, S., AND STÅLMARCK, G. 2000. Checking safety properties using induction and a SAT-solver. In *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, Texas, USA, November 1-3, 2000, Proceedings*, W. A. Hunt and S. D. Johnson, Eds., *Lecture Notes in Computer Science* vol. 1954 (2000). Springer-Verlag, 108–125.

VAN BEEK, P. AND CHEN, X. 1999. CPlan: A constraint programming approach to planning. In *Proceedings of the 16th National Conference on Artificial Intelligence (AAAI-99) and the 11th Conference on Innovative Applications of Artificial Intelligence (IAAI-99)* (1999). AAAI Press, 585–590.

WOLFMAN, S. A. AND WELD, D. S. 1999. The LPSAT engine & its application to resource planning. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, T. Dean, Ed., vol. I (1999). Morgan Kaufmann Publishers, 310–315.

ZARPAS, E. 2004. Simple yet efficient improvements of SAT based bounded model checking. In *Formal Methods in Computer-Aided Design: 5th International Confrence, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004. Proceedings*, A. J. Hu and A. K. Martin, Eds., Lecture Notes in Computer Science no. 3312 (2004). Springer-Verlag, 174–185.

APPENDIX

instance	len	val	∃-step	process	∀-step	∀-step l.
gripper-2	5	F	0.01 $^{0.01}_{0.01}$			
gripper-2	6	T	0.01 $^{0.01}_{0.01}$			
gripper-2	10	F		0.14 $^{0.13}_{0.15}$	0.08 $^{0.08}_{0.09}$	0.12 $^{0.12}_{0.13}$
gripper-2	11	T		0.04 $^{0.03}_{0.04}$	0.02 $^{0.01}_{0.02}$	0.05 $^{0.04}_{0.05}$
gripper-3	7	F	0.23 $^{0.23}_{0.24}$			
gripper-3	8	T	0.17 $^{0.16}_{0.18}$			
gripper-3	14	F		9.39 $^{8.43}_{10.47}$	3.91 $^{3.48}_{4.35}$	8.84 $^{7.84}_{9.93}$
gripper-3	15	T		1.72 $^{1.18}_{2.34}$	0.32 $^{0.19}_{0.47}$	0.69 $^{0.36}_{1.08}$
gripper-4	9	F	12.87 $^{11.61}_{14.26}$			
gripper-4	10	T	0.85 $^{0.70}_{1.02}$			
gripper-4	16	F		–	–	–
gripper-4	17	?		–	–	–
gripper-4	18	?		–	–	–
gripper-4	19	T		–	–	–

Table X. Runtimes of Gripper problems

instance	len	val	∃-step	process	∀-step	∀-step l.
satel-14	4	F	9.43 $^{8.81}_{10.12}$			
satel-14	5	T	1.79 $^{1.66}_{1.91}$			
satel-14	7	F		38.20 $^{36.42}_{40.13}$	29.59 $^{28.17}_{31.12}$	30.95 $^{29.59}_{32.37}$
satel-14	8	T		6.20 $^{5.83}_{6.61}$	4.38 $^{4.06}_{4.73}$	5.82 $^{5.31}_{6.37}$
satel-15	4	F	10.44 $^{9.36}_{11.66}$			
satel-15	5	T	1.60 $^{1.45}_{1.75}$			
satel-15	7	F		33.04 $^{30.92}_{35.37}$	26.58 $^{25.32}_{27.87}$	28.11 $^{26.72}_{29.59}$
satel-15	8	T		7.53 $^{7.17}_{7.90}$	4.83 $^{4.58}_{5.10}$	6.23 $^{5.93}_{6.55}$
satel-16	3	F	1.73 $^{1.54}_{1.93}$			
satel-16	4	T	3.36 $^{3.16}_{3.57}$			
satel-16	5	F		20.34 $^{18.68}_{22.04}$	8.80 $^{8.10}_{9.53}$	20.09 $^{18.61}_{21.74}$
satel-16	6	?		–	–	–
satel-16	7	T		8.87 $^{8.21}_{9.55}$	7.88 $^{7.42}_{8.39}$	7.81 $^{7.35}_{8.29}$
satel-17	3	F	0.28 $^{0.25}_{0.30}$			
satel-17	4	T	2.85 $^{2.81}_{2.90}$			
satel-17	5	F		2.74 $^{2.46}_{3.08}$	1.45 $^{1.32}_{1.63}$	1.72 $^{1.66}_{1.78}$
satel-17	6	T		3.46 $^{3.22}_{3.71}$	2.22 $^{2.10}_{2.35}$	2.53 $^{2.37}_{2.69}$
satel-18	4	F	0.07 $^{0.07}_{0.07}$			
satel-18	5	T	0.22 $^{0.20}_{0.24}$			
satel-18	7	F		0.60 $^{0.57}_{0.63}$	0.30 $^{0.29}_{0.31}$	0.54 $^{0.52}_{0.57}$
satel-18	8	T		1.18 $^{1.08}_{1.27}$	0.54 $^{0.49}_{0.58}$	0.86 $^{0.78}_{0.93}$

Table XI. Runtimes of Satellite problems

instance	len	val	∃-step	process	∀-step	∀-step l.
block-12-1	33	F	0.06 $^{0.05}_{0.06}$	0.17 $^{0.16}_{0.18}$	0.06 $^{0.06}_{0.06}$	0.16 $^{0.16}_{0.17}$
block-12-1	34	T	0.05 $^{0.04}_{0.05}$	0.36 $^{0.35}_{0.37}$	0.05 $^{0.05}_{0.05}$	0.19 $^{0.18}_{0.20}$
block-14-1	35	F	0.34 $^{0.34}_{0.35}$	1.45 $^{1.38}_{1.53}$	0.35 $^{0.34}_{0.35}$	1.01 $^{0.98}_{1.05}$
block-14-1	36	T	0.14 $^{0.12}_{0.15}$	1.18 $^{1.10}_{1.26}$	0.12 $^{0.11}_{0.14}$	0.50 $^{0.46}_{0.53}$
block-16-1	53	F	0.67 $^{0.65}_{0.69}$	3.82 $^{3.66}_{4.00}$	0.65 $^{0.63}_{0.68}$	1.77 $^{1.69}_{1.85}$
block-16-1	54	T	0.35 $^{0.33}_{0.37}$	4.95 $^{4.63}_{5.30}$	0.38 $^{0.36}_{0.40}$	1.86 $^{1.76}_{1.98}$
block-18-0	57	F	1.91 $^{1.85}_{1.98}$	15.20 $^{14.32}_{16.11}$	2.29 $^{2.22}_{2.36}$	6.56 $^{6.33}_{6.80}$
block-18-0	58	T	0.94 $^{0.87}_{1.01}$	8.04 $^{7.41}_{8.67}$	1.07 $^{0.98}_{1.17}$	3.42 $^{3.19}_{3.66}$
block-20-0	59	F	2.49 $^{2.37}_{2.61}$	8.34 $^{8.04}_{8.66}$	2.57 $^{2.43}_{2.74}$	5.37 $^{5.09}_{5.66}$
block-20-0	60	T	1.86 $^{1.79}_{1.92}$	9.55 $^{9.25}_{9.87}$	1.80 $^{1.74}_{1.85}$	4.93 $^{4.66}_{5.21}$
block-22-0	71	F	38.15 $^{36.82}_{39.48}$	–	38.49 $^{37.28}_{39.76}$	51.64 $^{49.58}_{53.78}$
block-22-0	72	T	14.34 $^{12.88}_{15.82}$	–	14.32 $^{13.03}_{15.66}$	26.72 $^{24.83}_{28.64}$

Table XII. Runtimes of Blocks World problems

instance	len	val	∃-step	process	∀-step	∀-step l.
driver-2-3-6b	4	F	0.01 $^{0.01}_{0.01}$			
driver-2-3-6b	5	T	0.01 $^{0.01}_{0.01}$			
driver-2-3-6b	6	F		0.04 $^{0.04}_{0.04}$	0.01 $^{0.01}_{0.01}$	0.02 $^{0.02}_{0.02}$
driver-2-3-6b	7	T		0.09 $^{0.08}_{0.09}$	0.03 $^{0.02}_{0.03}$	0.04 $^{0.04}_{0.05}$
driver-2-3-6c	6	F	0.01 $^{0.01}_{0.01}$			
driver-2-3-6c	7	T	0.01 $^{0.01}_{0.01}$			
driver-2-3-6c	8	F		0.03 $^{0.03}_{0.03}$	0.03 $^{0.03}_{0.03}$	0.03 $^{0.03}_{0.04}$
driver-2-3-6c	9	T		0.24 $^{0.22}_{0.27}$	0.10 $^{0.09}_{0.11}$	0.14 $^{0.13}_{0.16}$
driver-2-3-6d	12	F	0.44 $^{0.42}_{0.46}$			
driver-2-3-6d	13	T	0.63 $^{0.57}_{0.69}$			
driver-2-3-6d	15	F		34.14 $^{32.82}_{35.45}$	19.09 $^{18.23}_{20.00}$	26.27 $^{25.27}_{27.27}$
driver-2-3-6d	16	T		17.79 $^{15.95}_{19.67}$	8.04 $^{7.12}_{8.97}$	9.59 $^{8.30}_{11.00}$
driver-2-3-6e	7	F	0.01 $^{0.01}_{0.02}$			
driver-2-3-6e	8	T	0.04 $^{0.04}_{0.04}$			
driver-2-3-6e	11	F		2.14 $^{2.04}_{2.24}$	1.13 $^{1.08}_{1.19}$	1.55 $^{1.47}_{1.62}$
driver-2-3-6e	12	T		2.54 $^{2.29}_{2.80}$	1.27 $^{1.10}_{1.45}$	1.25 $^{1.09}_{1.41}$
driver-3-3-6b	8	F	0.16 $^{0.15}_{0.17}$			
driver-3-3-6b	9	T	0.08 $^{0.07}_{0.09}$			
driver-3-3-6b	10	F		2.15 $^{1.99}_{2.31}$	0.82 $^{0.77}_{0.87}$	1.40 $^{1.31}_{1.51}$
driver-3-3-6b	11	T		3.26 $^{2.77}_{3.83}$	1.07 $^{0.90}_{1.26}$	1.43 $^{1.21}_{1.69}$
driver-4-4-8	8	F	0.14 $^{0.13}_{0.15}$			
driver-4-4-8	9	T	0.15 $^{0.13}_{0.16}$			
driver-4-4-8	10	F		4.68 $^{4.49}_{4.87}$	1.30 $^{1.26}_{1.33}$	2.84 $^{2.75}_{2.94}$
driver-4-4-8	11	T		23.69 $^{21.93}_{25.60}$	5.92 $^{5.35}_{6.49}$	13.08 $^{11.94}_{14.26}$

Table XIII. Runtimes of DriverLog problems

instance	len	val	∃-step	process	∀-step	∀-step l.
sched-10-0	6	F	0.01 $^{0.01}_{0.01}$	0.01 $^{0.01}_{0.01}$	0.01 $^{0.01}_{0.01}$	0.01 $^{0.01}_{0.01}$
sched-10-0	7	T	0.01 $^{0.01}_{0.01}$	0.07 $^{0.05}_{0.10}$	0.01 $^{0.01}_{0.01}$	0.01 $^{0.01}_{0.01}$
sched-15-0	8	F	8.74 $^{7.40}_{10.17}$	14.57 $^{13.27}_{15.88}$	3.44 $^{2.86}_{4.10}$	11.52 $^{10.21}_{12.85}$
sched-15-0	9	T	0.16 $^{0.12}_{0.22}$	0.23 $^{0.19}_{0.27}$	0.14 $^{0.11}_{0.16}$	0.36 $^{0.27}_{0.44}$
sched-20-0	8	F	1.11 $^{1.06}_{1.16}$	1.42 $^{1.37}_{1.47}$	0.46 $^{0.44}_{0.48}$	1.30 $^{1.25}_{1.35}$
sched-20-0	9	T	0.14 $^{0.11}_{0.18}$	0.24 $^{0.21}_{0.28}$	0.19 $^{0.19}_{0.20}$	0.10 $^{0.09}_{0.11}$
sched-25-0	8	F	7.85 $^{6.87}_{8.96}$	15.47 $^{14.61}_{16.37}$	2.14 $^{1.96}_{2.35}$	8.53 $^{7.57}_{9.56}$
sched-25-0	9	T	0.29 $^{0.23}_{0.35}$	0.68 $^{0.55}_{0.82}$	0.19 $^{0.18}_{0.21}$	0.69 $^{0.56}_{0.84}$
sched-30-0	10	F	–	–	8.12 $^{5.93}_{10.45}$	–
sched-30-0	11	T	1.05 $^{0.78}_{1.36}$	2.63 $^{2.22}_{3.04}$	1.07 $^{0.88}_{1.29}$	0.90 $^{0.65}_{1.19}$
sched-35-0	10	F	26.22 $^{24.78}_{27.71}$	34.35 $^{32.89}_{35.87}$	10.26 $^{9.72}_{10.80}$	30.09 $^{28.35}_{31.88}$
sched-35-0	13	T	3.43 $^{2.66}_{4.37}$	3.53 $^{3.02}_{4.09}$	3.86 $^{3.15}_{4.68}$	3.14 $^{2.49}_{3.96}$

Table XIV. Runtimes of Schedule problems

instance	len	val	∃-step	process	∀-step	∀-step l.
zeno-3-7b	3	F	0.01 $^{0.01}_{0.01}$			
zeno-3-7b	4	T	0.01 $^{0.01}_{0.01}$			
zeno-3-7b	5	F		0.10 $^{0.10}_{0.10}$	0.02 $^{0.02}_{0.02}$	0.05 $^{0.05}_{0.06}$
zeno-3-7b	6	T		0.11 $^{0.10}_{0.12}$	0.02 $^{0.02}_{0.02}$	0.06 $^{0.05}_{0.06}$
zeno-3-8	3	F	0.01 $^{0.01}_{0.01}$			
zeno-3-8	4	T	0.01 $^{0.01}_{0.01}$			
zeno-3-8	5	F		0.08 $^{0.08}_{0.09}$	0.02 $^{0.02}_{0.02}$	0.05 $^{0.05}_{0.05}$
zeno-3-8	6	T		0.49 $^{0.45}_{0.53}$	0.06 $^{0.06}_{0.07}$	0.30 $^{0.27}_{0.33}$
zeno-3-8b	3	F	0.01 $^{0.01}_{0.01}$			
zeno-3-8b	4	T	0.02 $^{0.01}_{0.02}$			
zeno-3-8b	5	F		0.17 $^{0.16}_{0.17}$	0.03 $^{0.02}_{0.03}$	0.11 $^{0.11}_{0.12}$
zeno-3-8b	6	T		0.54 $^{0.47}_{0.61}$	0.16 $^{0.16}_{0.16}$	0.31 $^{0.27}_{0.36}$
zeno-3-10	4	F	0.05 $^{0.05}_{0.05}$			
zeno-3-10	5	T	0.02 $^{0.02}_{0.02}$			
zeno-3-10	6	F		1.77 $^{1.71}_{1.84}$	0.51 $^{0.50}_{0.51}$	1.17 $^{1.13}_{1.21}$
zeno-3-10	7	T		2.68 $^{2.43}_{2.95}$	0.76 $^{0.68}_{0.86}$	1.79 $^{1.57}_{2.01}$
zeno-5-10	3	F	0.10 $^{0.10}_{0.10}$			
zeno-5-10	4	T	0.23 $^{0.19}_{0.28}$			
zeno-5-10	5	F		2.23 $^{2.13}_{2.33}$	1.03 $^{1.03}_{1.04}$	1.47 $^{1.41}_{1.53}$
zeno-5-10	6	T		9.34 $^{8.78}_{9.89}$	3.17 $^{2.79}_{3.57}$	6.53 $^{6.04}_{7.08}$
zeno-5-15	5	F	–			
zeno-5-15	6	T	21.34 $^{17.83}_{25.10}$			
zeno-5-15	5	F		3.52 $^{3.30}_{3.75}$	1.76 $^{1.75}_{1.76}$	2.32 $^{2.18}_{2.48}$
zeno-5-15	6	?		–	–	–
zeno-5-15	7	T		–	39.34 $^{35.65}_{43.34}$	–

Table XV. Runtimes of ZenoTravel problems

instance	len	val	∃-step	process	∀-step	∀-step l.
depot-13-5646	7	F	0.01 $^{0.01}_{0.01}$			
depot-13-5646	8	T	0.01 $^{0.01}_{0.01}$			
depot-13-5646	8	F		0.02 $^{0.02}_{0.02}$	0.01 $^{0.01}_{0.01}$	0.01 $^{0.01}_{0.01}$
depot-13-5646	9	T		0.27 $^{0.26}_{0.29}$	0.04 $^{0.04}_{0.05}$	0.08 $^{0.08}_{0.09}$
depot-14-7654	9	F	0.05 $^{0.05}_{0.06}$			
depot-14-7654	10	T	0.10 $^{0.09}_{0.11}$			
depot-14-7654	11	F		3.07 $^{2.95}_{3.19}$	1.41 $^{1.34}_{1.48}$	2.17 $^{2.07}_{2.29}$
depot-14-7654	12	T		8.18 $^{7.55}_{8.82}$	3.48 $^{3.19}_{3.78}$	4.26 $^{3.87}_{4.66}$
depot-16-4398	7	F	0.01 $^{0.01}_{0.01}$			
depot-16-4398	8	T	0.01 $^{0.01}_{0.01}$			
depot-16-4398	7	F		0.03 $^{0.03}_{0.03}$	0.01 $^{0.01}_{0.01}$	0.02 $^{0.01}_{0.02}$
depot-16-4398	8	T		0.43 $^{0.41}_{0.46}$	0.07 $^{0.06}_{0.07}$	0.12 $^{0.11}_{0.13}$
depot-17-6587	5	F	0.01 $^{0.01}_{0.01}$			
depot-17-6587	6	T	0.01 $^{0.01}_{0.01}$			
depot-17-6587	6	F		0.24 $^{0.23}_{0.26}$	0.02 $^{0.02}_{0.02}$	0.13 $^{0.11}_{0.14}$
depot-17-6587	7	T		0.69 $^{0.65}_{0.74}$	0.03 $^{0.03}_{0.03}$	0.27 $^{0.25}_{0.29}$
depot-18-1916	11	F	0.29 $^{0.28}_{0.29}$			
depot-18-1916	12	T	5.80 $^{5.02}_{6.61}$			
depot-18-1916	11	F		1.12 $^{1.04}_{1.20}$	0.17 $^{0.16}_{0.17}$	0.51 $^{0.48}_{0.54}$
depot-18-1916	12	T		–	–	–

Table XVI. Runtimes of Depot problems

instance	len	val	∃-step	process	∀-step	∀-stepl.
freecell2-4	4	F	0.01 $^{0.01}_{0.01}$	0.01 $^{0.01}_{0.01}$	0.01 $^{0.01}_{0.01}$	0.01 $^{0.01}_{0.01}$
freecell2-4	5	T	0.01 $^{0.01}_{0.01}$	0.02 $^{0.01}_{0.02}$	0.01 $^{0.01}_{0.01}$	0.01 $^{0.01}_{0.01}$
freecell3-4	7	F	0.45 $^{0.43}_{0.47}$	0.77 $^{0.74}_{0.81}$	0.25 $^{0.25}_{0.25}$	0.53 $^{0.50}_{0.55}$
freecell3-4	8	T	0.13 $^{0.11}_{0.15}$	0.25 $^{0.22}_{0.28}$	0.18 $^{0.17}_{0.18}$	0.11 $^{0.10}_{0.13}$
freecell4-4	6	F	0.01 $^{0.01}_{0.01}$	0.01 $^{0.01}_{0.01}$	0.01 $^{0.01}_{0.01}$	0.01 $^{0.01}_{0.01}$
freecell4-4	7	T	0.05 $^{0.04}_{0.05}$	0.12 $^{0.11}_{0.13}$	0.02 $^{0.02}_{0.02}$	0.08 $^{0.07}_{0.08}$
freecell5-4	12	F	13.60 $^{12.94}_{14.29}$	17.34 $^{16.54}_{18.14}$	6.75 $^{6.39}_{7.17}$	9.19 $^{8.84}_{9.55}$
freecell5-4	13	T	59.57 $^{52.44}_{66.68}$	63.78 $^{57.28}_{70.33}$	35.70 $^{32.55}_{39.06}$	53.59 $^{47.44}_{60.17}$

Table XVII. Runtimes of FreeCell problems