Parallel Encodings of Classical Planning as Satisfiability

Jussi Rintanen
Albert-Ludwigs-Universität Freiburg
Institut für Informatik
Georges-Köhler-Allee
79110 Freiburg im Breisgau
Germany

Keijo Heljanko
University of Stuttgart
Institute for Formal Methods
in Computer Science
Universitätsstrasse 38, 70569 Stuttgart
Germany

Ilkka Niemelä
Helsinki University of Technology
Laboratory for Theoretical Computer Science
P. O. Box 5400, FIN-02015 HUT
Finland

Abstract

We consider a number of semantics for plans with parallel operator application. The standard semantics most often used in earlier work requires that parallel operators are independent, and can therefore executed in any order. We consider a more relaxed definition of parallel plans, first proposed by Dimopoulos et al., as well as normal forms for parallel plans that require every operator executed as early as possible, first proposed outside planning in connection with Petri nets. We formalize the semantics of parallel plans emerging in this setting, propose effective translations of these semantics to the classical propositional logic, and show that they yield an approach to classical planning that is often much more efficient than the existing SAT-based approaches.

Introduction

Finding paths in transition systems – as in model-checking (Biere *et al.* 1999) and AI planning (Kautz & Selman 1996) – is one of the important applications of satisfiability testing.

Efficiency of satisfiability algorithms in problem solving is strongly determined by details of the problem representation. In the case of AI planning, techniques that allow considering plans with a smaller number of time points – especially parallel plans (Kautz & Selman 1996; Blum & Furst 1997) – and thereby representations of the planning problem with a smaller number of propositional variables, as well as constraints for reducing the space of incomplete solutions – like invariants/mutexes (Kautz & Selman 1996; Blum & Furst 1997) – are of great importance in achieving efficient planning.

In this paper we address the encoding of parallel plans in the propositional logic. In the *state-based encoding* of deterministic AI planning (Kautz & Selman 1996), between two time points the simultaneous application of several operators is allowed provided that the operators are mutually non-interfering. This non-interference condition guarantees that any total ordering on the simultaneous operators is a valid execution and in all cases leads to the same state. The two benefits of this kind of parallel encodings are that, first, it is unnecessary to consider all possible orderings of a set of non-interfering operators, and second, less propositional variables are needed as the values of the state variables in the (implicit) intermediate states need not be represented, reducing the size of formulae.

Heljanko (2001) has applied a stricter semantics for parallel transitions/operators for deadlock detection of 1-safe Petri nets, called *process semantics* (as opposed to the standard semantics for parallelism, *step semantics*), which requires that all transitions take place as early as possible, that is, no transition could have taken place one time point earlier. Process semantics allows constraining the set of allowed transition sequences (plans), and has been shown to lead to efficiency gains on many types of problems in bounded model-checking (Heljanko 2001).

Our first goal in this paper is to present a process semantics for deterministic AI planning, consider its representation in encodings of planning in the propositional logic, and evaluate the possible efficiency gains induced by the stronger constraints on the sets of possible plans.

Another refinement to the idea of parallel plans was suggested by Dimopoulos et al. (1997). They pointed out that it is not necessary to require that all parallel operators are non-interfering, as long as there is a guarantee that the parallel operators can be executed in at least one order. Dimopoulos et al. discussed this idea in terms of blocks world and logistics problems encoded as nonmonotonic logic programs (Dimopoulos, Nebel, & Koehler 1997), and showed that efficiency gains can be obtained. This more relaxed semantics allows a further reduction in the number of time points for which the values of the state variables have to be represented explicitly. We call this the *1-linearization semantics*.

Our second goal is to provide a general formalization for 1-linearization semantics, to provide a general and efficient translation of 1-linearization semantics to the propositional logic, and to evaluate its computational behavior.

The structure of the paper is as follows. First in the next section we discuss those parts of the standard state-based encoding of AI planning in the classical propositional logic that are independent of the semantics for parallelism. Then we introduce the underlying ideas of the process semantics, formally define a process semantics for AI planning, and discusses the representation of process semantics in the propositional logic. This is followed by a section on the more relaxed 1-linearization semantics and its translation to the propositional logic. The experiments' section evaluates the advantages and disadvantages of the different semantics in terms of some planning problems. The last section concludes the paper by discussing future research directions.

Notation

Definition 1 *An* operator *on a set of state variables P is a triple $\langle p, e, c\rangle$ where*

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

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

$$e \cup \bigcup \{d | f \rhd d \in c, s \models f\}.$$

The operator is *applicable* in s if $s \models p$ and its active effects are consistent. We define $\text{app}_o(s) = s'$ to be the unique state that is obtained by applying the operator o in state s. The state s' is obtained from s by setting the operator's active effects true, and retaining the truth-values of state variables not mentioned in the active effects. For sequences $o_1; o_2; \ldots; o_n$ of operators we define $\text{app}_{o_1;o_2;\ldots;o_n}(s)$ as $\text{app}_{o_n}(\cdots \text{app}_{o_2}(\text{app}_{o_1}(s)) \cdots)$. For sets S of operators and states s we define $\text{app}_S(s)$: the result of simultaneously applying all operators $o \in S$. We require that $\text{app}_o(s)$ is defined for every $o \in S$, and the set of active effects of all operators in S is consistent. In later sections we impose further restrictions on operators that are applied simultaneously, because we want to interpret simultaneous application for example as applying operators in S in any order reaching the same state in all cases.

Planning as Satisfiability

The idea of representing planning as a propositional satisfiability problem was proposed by Kautz and Selman (1996). Several different encodings of planning have been presented (Kautz & Selman 1996), and in this section we present a variant of one of the more important ones which Kautz and Selman call the state-based encoding. The encodings of planning for the different semantics for parallelism differ only on a small number of axioms that restrict the simultaneous application of operators. Next we describe the common part of the encodings that is independent of the semantics for parallelism.

The state variables in a problem instance are $P = \{a^1, \ldots, a^n\}$. The operators in a problem instance are $O = \{o^1, \ldots, o^m\}$.

For a state variable a we have the propositional variable a_t that expresses the truth-value of a at time point t. Similarly, for an operator o we have the propositional variable o_t for expressing whether the operator is applied at time point t. For formulae ϕ, we denote the formula with all propositional variables subscripted with the index to a time point t by ϕ_t.

A formula is generated to answer the following question. Is there a transition sequence taking l time points that reaches a goal state satisfying G from the initial state?

The formula is conjunction of I_0, G_l and the formulae described below, instantiated for all $t \in \{0, \ldots, l-1\}$.

First, for every operator $o = \langle p, e, c\rangle, o \in O$ there are axioms for stating what the preconditions and effects of the operator are. The preconditions have to be true when the operator is applied.

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

If the operator is applied, then its (unconditional) effects are true at the next time point,

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

Here we view the effect e (a set of literals) as a conjunction of literals. The conditional effects are true whenever the antecedents of the conditionals are true in the preceding time point:

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

for every $f \rhd d \in c$.

Second, the values of state variables can be determined by the fact that no operator that could change them has been applied. They known as frame axioms for historical reasons. For every state variable a we have two formulae, one expressing the conditions for the change of a to false from true, and the other from true to false. The formulae are analogous, and we only describe the change from true to false, which is simply by

$$(a_t \wedge \neg a_{t+1}) \rightarrow ((o_t^1 \wedge \phi_t^1) \vee \cdots \vee (o_t^m \wedge \phi_t^m)) \tag{4}$$

where formula ϕ^i expresses the condition under which operator o^i changes a from true to false. So let $o^i = \langle p, e, c\rangle$. If a is a negative effect in e, then simply

$$\phi^i = \top.$$

Otherwise, the change takes place if one of the conditional effects is active. Let $f^1 \rhd d^1, \ldots, f^k \rhd d^k$ be the conditional effects with a as a negative effect in d^j. Here $k \geq 0$. Then

$$\phi^i = f^1 \vee \ldots \vee f^k.$$

For $k = 0$ the empty disjunction is the constant false $\bot$.

Finally, we need axioms for restricting the parallel application of operators: we will describe them in the next sections for each semantics of parallelism. The resulting formulae are satisfiable if and only if there is a transition sequence taking l time points that reaches a goal state from the unique initial state.

In addition to the above axioms which are necessary to guarantee that the set of satisfying assignments exactly corresponds to the set of plans with l time points, it is often useful to add further constraints that do not affect the set of satisfying assignments, but help in pruning the set of incomplete solutions need to be looked at, and thereby speed up plan search [1].

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 some restricted class of invariants that are efficient (polynomial time) to identify. There are efficient algorithms for finding many invariants that are clauses consisting of two literals (Rintanen 1998; Blum & Furst 1997). We simply include the formulae

$$l_t \vee l'_t \tag{5}$$

[1] There are further constraints that can speed plan search, like those derived from symmetries. They reduce the set of possible plans, but allow at least one plan whenever at least one plan exists.

for invariants $l \vee l'$. Notice that invariants are in many cases needed because the problem encodings produced from PDDL or similar input languages are often not very economical in terms of the number of state variables. For example, typical benchmarks encode what is essentially an n-valued state variable by n Boolean state variables, instead of $\lceil \log_2 n \rceil$ Boolean state variables.

Step Semantics

For giving a definition of parallel plans under step semantics, we need to define when operators interfere in a way that makes their simultaneous application unwanted.

Definition 2 (Interference) *Operators $o_1 = \langle p_1, e_1, c_1 \rangle$ and $o_2 = \langle p_2, e_2, c_2 \rangle$* interfere in state s *(a valuation of all state variables) if*

1. $s \models p_1 \wedge p_2$,
2. *the set $e_1 \cup \bigcup\{d | f \rhd d \in c_1, s \models f\} \cup e_2 \cup \bigcup\{d | f \rhd d \in c_2, s \models f\}$ is consistent, and*
3. *the operators are not applicable in both orders, or applying them in different orders leads to different results:*
   (a) $app_{o_1}(s) \not\models p_2$,
   (b) $app_{o_2}(s) \not\models p_1$, *or*
   (c) *active effects of o_2 are different in s and in $app_{o_1}(s)$ or active effects of o_1 are different in s and in $app_{o_2}(s)$.*

The first two conditions are the *applicability conditions* for parallel operators: preconditions have to be satisfied and the effects may not contradict each other. The third condition says that interference is impossibility to interpret parallel application as application in any order, o_1 followed by o_2, or o_2 followed by o_1, leading to the same state in both cases: one execution order is impossible, or the resulting states are different.

The conditions guarantee that any two consecutive non-interfering operators may be interchanged without affecting the states that are visited thereafter.

Notice that some earlier work – contrary to the above definition – has considered contradicting preconditions or effects as one form of interference. We do not consider this as interference, because simultaneous application of the operators with contradicting preconditions or effects is prevented by the precondition and effect axioms 1, 2 and 3.

Definition 3 (Step plans) *For a set of operators O and an initial state I,* a plan *is a sequence $P = S_1, \ldots, S_l$ of sets of operators such that there is a sequence of states $s_0, \ldots, s_l$ (the execution of P) such that*

1. *the operators in S_i are applicable in s_{i-1} for all $i \in \{1, \ldots, l\}$, that is, $s_{i-1} \models p$ for every $\langle p, e, c \rangle \in S_i$,*
2. *the set $\bigcup_{\langle p,e,c \rangle \in S_i} \{e \cup \bigcup\{d | f \rhd d \in c, s_{i-1} \models f\}$ is consistent for every $i \in \{1, \ldots, l\}$,*
3. $s_0 = I$,
4. $s_i = app_{S_i}(s_{i-1})$ *for $i \in \{1, \ldots, l\}$, and*
5. *for all $i \in \{1, \ldots, l\}$ and $o, o' \in S_i$, o and o' do not interfere in $app_S(s_{i-1})$ for any $S \subseteq S_i \backslash \{o, o'\}$.*

Theorem 4 *Let $P_1 = S_1, \ldots, S_k, \ldots, S_l$ be a step plan with initial state s_0. Let $P_2 = S_1, \ldots, S_k^0, S_k^1, \ldots, S_l$ be a step plan that is obtained from P_1 by splitting the step S_k into two steps S_k^0 and S_k^1 such that $S_k = S_k^0 \cup S_k^1$.*

If $s_0, \ldots, s_{k-1}, s_k, \ldots, s_l$ is the execution of P_1, then $s_0, \ldots, s_{k-1}, s_k', s_k, \ldots, s_l$ for some state s_k' is the execution of P_2.

Corollary 5 *Any sequence of operators that is a total ordering of the operators in a step plan is a plan and has the same terminal state.*

Proof: One can repeatedly split non-singleton steps until all steps are singleton. □

Encoding in the Propositional Logic

The application of two operators must be prevented in states in which they interfere. In practice, most work on satisfiability planning has tested a counterpart of Condition 3 in Definition 2 by a simple syntactic test. For example, prevent the simultaneous application of two operators whenever one of the state variables affected by one operator occur in the precondition or in the antecedents of conditional effects of the other with a different polarity. Then for any such pair of operators o and o', include the axioms

$$\neg o_t \vee \neg o_t'. \tag{6}$$

Notice that for STRIPS operators, that is operators with unconditional effects only and with a precondition that is a conjunction of literals, these constraints allow all the parallelism that is possible.

Process Semantics

The idea of process semantics is that operators are always applied as early as possible. Assume that no two operators in set S interfere, have no contradicting effects, and all are applicable in state s. 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 step semantics the number of equivalent plans on two time points is $2^{|S|}$. Process semantics would in this case say that no operator that is applicable at 0 may be applied later than at 0. So, instead of the $2^{|S|}$ plans as in step semantics, under process semantics there is only one plan.

The important property of process semantics is that even though the additional conditions reduce the number of acceptable plans, whenever there exists a plan with t time steps under step semantics, there is also a plan with t time steps under process semantics. The plan satisfying the process conditions is obtained from the step plan by repeatedly moving operators violating the conditions one step earlier.

Encoding in the Propositional Logic

The encoding of process semantics is an extension of the encoding of step semantics, so we take all the axioms from the preceding section, and have further axioms specific to process semantics.

The axioms for process semantics allow the application of operator o at time $t+1$ only if one of the operators at time t interferes with o according to Definition 2, the effects of o contradict with the effects of an operator at time point t, or o is not enabled at time t.

Other possibilities, like operator o disabling or changing the active effects of operators at $t+1$ do not have to be tested separately, because this is already forbidden by the axioms we have from step semantics.

So let $\{o^1, \ldots, o^n\}$ be the operators interfering with o, having conflicting effects with o, or having possibly enabled o. Then we have the following axioms for guaranteeing that the application of o is not unnecessarily delayed.

$$o_{t+1} \rightarrow (o_t^1 \vee o_t^2 \vee \cdots \vee o_t^n)$$

These disjunctions may be long. Instead of using them for implementing the process semantics exactly, it may be useful to use only the shortest of these constraints, as they are most likely to help speeding up plan search. In the experiments reported later, we allowed only those constraints consisting of 6 literals or less. We tried full process semantics, but the high number of long disjunctions led to poor performance.

1-Linearization Semantics

Dimopoulos et al. (1997) adapted the idea of satisfiability planning for answer set programming, and presented an interesting idea. The requirement that parallel operators are executable in any order can be relaxed: only require that *one* ordering is executable. They called this idea *post-serializability*, and showed how to transform operators for blocks world problems to make them post-serializable. The resulting nonmonotonic logic programs were shown to be more efficient due to shorter parallel plan length. Rintanen (1998) implemented this idea in a constraint-based planner and Cayrol et al. in an implementation of the GraphPlan algorithm (Cayrol, Régnier, & Vidal 2001), but otherwise it has received surprisingly little attention, in particular, there have been no attempts to utilize it in satisfiability planning.

We will present a semantics and general-purpose domain-independent translations of this more relaxed semantics to propositional logic. Our approach does not require transforming the problem. Instead, we synthesize contraints that guarantee that sets of operators applied simultaneously can be ordered to an executable plan.

Definition 6 (1-linearization plans) *For a set of operators O and an initial state I,* a 1-linearization plan *is a sequence $P = S_1, \ldots, S_l$ of sets of operators such that there is a sequence of states $s_0, \ldots, s_l$ (the execution of P) such that*

1. *the operators in S_i are applicable in s_{i-1} for all $i \in \{1, \ldots, l\}$, that is, $s_{i-1} \models p$ for every $\langle p, e, c\rangle \in S_i$,*
2. *the set $\bigcup_{\langle p,e,c\rangle \in S_i}(e \cup \bigcup\{d | f \rhd d \in c, s_{i-1} \models f\})$ is consistent for every $i \in \{1, \ldots, l\}$,*
3. *$s_0 = I$,*
4. *$s_i = app_{S_i}(s_{i-1})$ for $i \in \{1, \ldots, l\}$, and*
5. *for all $i \in \{1, \ldots, l\}$, there is a total ordering $o_1 < o_2 < \ldots < o_n$ of S_i such that for all operators $o_j = \langle p_j, e_j, c_j\rangle \in S_i$*
   (a) $app_{o_1;o_2;\ldots;o_{j-1}}(s_{i-1}) \models p_j$, and
   (b) for all $f \rhd d \in c_j$, $s_{i-1} \models f$ if and only if $app_{o_1;o_2;\ldots;o_{j-1}}(s_{i-1}) \models f$.

The difference to step semantics is that we have replaced the non-interference condition with a weaker condition. From an implementations point of view the main difficulty here is finding appropriate total orderings $<$.

Encoding in the Propositional Logic

The problem to be solved by the encoding of 1-linearization semantics in the propositional logic is preventing the simultaneous application of a set of operators that cannot be applied sequantially in any order. Given the precondition and effect axioms (1), (2) and (3), the problem is to guarantee that there is a total ordering of the operators so that no operator application disables the operators to be applied later, and no operator application changes the set of active (conditional) effects of later operators.

Essentially, this is a question of topologically sorting the set of operators with respect to the disables-or-affects relation between the operators. Dimopoulos et al. (1997) defined the *preconditions-effects* graph for determining when such a total ordering exists for sets of STRIPS operators. The problem of using this graph directly as a basis of SAT encoding is that acyclicity testing seems to require a cubic number of clauses, and this makes it rather impractical for all but the smallest problem instances.

We define a new class of graphs that we call *disabling graphs* in order to provide compact and effective encodings of 1-linearization semantics in the propositional logic. By means of these graphs we can identify sets of operators that might not be possible to execute in any total ordering.

The motivation for using disabling graphs is the following. Any set S of operators that could be applied simultaneously ("could be applied" means that the operators are all applicable in one state and the effects do not conflict) and cannot be totally ordered to an executable operator sequence, and is a set-inclusion minimal set having the preceding properties, is a subset of a strong component of the disabling graph.

Definition 7 (Disabling graph) *A graph $\langle N, E\rangle$ is a* disabling graph *of a set of operators O if $N = O$ is the set of nodes, and E is a set of directed edges so that $\langle o_1, o_2\rangle \in E$ if for $o_1 = \langle p_1, e_1, c_1\rangle$ and $o_2 = \langle p_2, e_2, c_2\rangle$ there is at least one state s (valuation of all state variables)[2] such that*

1. *$s \models p_1 \wedge p_2$, and*
2. *$F_1 \cup F_2$ is consistent where $F_1 = e_1 \cup \bigcup\{d | f \rhd d \in c_1, s \models f\}$ and $F_2 = e_2 \cup \bigcup\{d | f \rhd d \in c_2, s \models f\}$, and*

[2]Clearly, this could be restricted to states that are reachable from the initial state, but testing reachability is PSPACE-hard. Instead, one can use some subclass of invariants identifiable in polynomial time to ignore some of the unreachable states, like we have done in our implementation of disabling graphs.

3. *applying o_1 may make o_2 inapplicable or may change the active effects of o_2:*
   (a) *$app_{o_1}(s) \not\models p_2$, or*
   (b) *there is $f \rhd d \in c_2$ such that either $s \models f$ and $app_{o_1}(s) \not\models f$, or $s \not\models f$ and $app_{o_1}(s) \models f$.*

Notice that 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. There is exactly one set-inclusion minimal disabling graph, but the use of non-minimal disabling graphs does not compromise the correctness of our translations, it may just make them less efficient or lead to plans with more time points. Non-minimal graphs may be useful because they may be much cheaper to compute than the minimal ones. Testing the condition for having an edge between o_1 and o_2 in the minimal disabling graph is clearly NP-hard. For operators of very simple syntactic form, like STRIPS operators, computing minimal disabling graphs is polynomial time.

Disabling graphs have less edges than precondition-effect graphs because there are never edges between operators that are not simultaneously applicable. Consequently, disabling graphs have smaller strong components, allowing much more effective handling of parallelism.

So, now we have an effective means of identifying which sets of simultaneous operators might not be possible to order to a valid plan, and as importantly, which operators are not problematic in this respect. The strong components of the disabling graph can be efficiently computed by the strong components algorithm of Tarjan (1972).

Interestingly, disabling graphs often have very small strong components. For example, in the well-known logistics problems all the strong components have 1 or $n + 1$ operators, where n is the number of airplanes[3].

This means that encoding the constraints that guarantee that simultaneous operators indeed can be linearized will be rather efficient, as only cycles of rather small length have to be considered. Notice that operators in different strong components cannot be part of the same cycle, and therefore no constraints on their simultaneous application are needed. Hence when every strong component has cardinality 1, no contraints whatsoever are needed.

Next we discuss three strategies for synthesizing constraints that guarantee that simultaneous operators can be ordered to a valid totally ordered plan.

General Solution

A general solution for guaranteeing that the intersection of one SCC and the set of operators applied at a given time point do not form a cycle is to exactly test this. The encoding we present here guarantees the maximum possible parallelism, but it is very expensive in terms of the size of the resulting formulae.

Let operators o^i and $o^{i'}$ belong to the same SCC of the disabling graph and let there be at least one state in which o^i disables or changes the active effects of $o^{i'}$. Let $\phi^{i,i'}$ be a formula such that o^i disables $o^{i'}$ or changes its set of active effects only in states that satisfy $\phi^{i,i'}$. Again, for guaranteeing maximal parallelism, $\phi^{i,i'}$ should be false in states where o_i does not disable nor affect the effects of $o^{i'}$, but $\phi^{i,i'}$ may be weaker without sacrificing correctness, at the cost of reduced parallelism. One strategy is to always use $\phi^{i,i'} = \top$, which is of course the best in terms of encoding size.

We introduce auxiliary propositions $c^{i,j}$ for all operators with indices i and j, indicating that there is a set of applied operators $o^i, o^1, o^2, \ldots, o^n, o^j$ such that every operator disables or changes the effects of its immediate successor in the sequence. Then we produce the following formulae, for all i, i' and j such that $i \neq i' \neq j \neq i$.

$$(o^i_t \wedge \phi^{i,i'}_t \wedge c^{i',j}_t) \rightarrow c^{i,j}_t$$

Further we have formulae

$$\neg(o^i_t \wedge \phi^{i,i'}_t \wedge c^{i',i}_t)$$

for preventing the completion of a cycle.

The size of the encoding is cubic in the number of operators in an SCC, and the number of new propositional variables is quadratic in the number of operators in an SCC. Some problems have SCCs of dozens or hundreds of operators, and this $O(n^3)$ means thousands or millions of formulae, which makes this encoding in general infeasible.

Fixed Ordering

The simplest and possibly most effective encoding does not allow all the parallelism that would be possible by the preceding encoding, but it minimizes the formula size. With this encoding, the number of constraints on parallel application *is smaller* than with the less permissive step semantics, as the set of contraints on parallelism is a subset of the contraints for step semantics. One therefore gets two benefits simultaneously, potentially 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 disallow parallel application of two operators o_i and o_j such that o_i may disable or change the effects of o_j only if $i < j$. Hence, in comparison to step semantics, part of the parallelism axioms on operators within one SCC are just left out. The total reduction in the number of constraints in comparison to step semantics can be significant because none of the inter-SCC parallelism contraints are needed.

This is the encoding we have very successfully applied to a wide range of planning problems, as discussed in the experiments' section. Clearly, how the fixed ordering is selected could have a big effect on planning efficiency. In our experiments we always ordered the operators exactly in the order they came out of our PDDL front-end. To maximize parallelism, better orderings could presumably be produced by some sophisticated heuristic method.

Enumeration of Cycles

A third encoding that for small SCCs may be a more feasible alternative to the first encoding, would be to explicitly enumerate all possible cycles in an SCC containing operators C

[3]The refinement to disabling graphs involving invariants and mentioned in the previous footnote makes all SCCs for the logistics problems singleton sets.

such that there is no set $C' \subset C$ that forms a cycle, and then use the clause

$$\bigvee\{\neg o_t | o \in C\}$$

to prevent the cycle from emerging during planning. For small SCCs the number of cycles may be smaller than the worst-case quadratic number of constraints needed in the preceding encoding. And unlike in the general encoding, no new propositions are needed.

Experiments

We evaluated the different semantics on a number of benchmarks from the AIPS planning competitions. In addition to the runtimes reported here, we ran further benchmarks from the competitions, the Freecell, Schedule, Mystery and Movie benchmarks, but do not report results because of lack of space. On Freecell and Schedule 1-linearization does not decrease plan length, and runtimes are comparable to step semantics. Process semantics fares worse than step semantics on Schedule. Mystery is trivial for 1-linearization semantics. Movie is trivial for all.

In Tables 1, 2, 3, 4 and 5 we report the name of the problem instance, and the runtimes under the three semantics for the formulae corresponding to the highest number(s) of time points not having a plan (truth value F), and the first satisfiable formula corresponding to a plan (truth value T). The runtimes for the 1-linearization semantics are reported on their own line because its shortest plan lengths differ from the other semantics.

The experiments were run on a 3.6 GHz Intel Xeon processor with a 512 KB internal cache and the Siege SAT solver version 3 by Lawrence Ryan of the Simon Fraser University. This solver is often much faster than for example zChaff. Because Siege randomizes some of the decisions it makes, the runtimes on a given formula vary across executions. We ran Siege 40 times on each formula and report the average. When not all of the runs finished within a time limit of 3 to 4 minutes (we terminated every 60 seconds those processes that had consumed over 180 seconds of CPU), we report the average time t of the finished runs as $> n$. This then very imprecisely means that the average runtime on Siege is somewhat higher than n seconds. When none of the runs finished within the time limit, we indicate this with a dash $-$.

Notice that problems with a bigger index are in general not necessarily more difficult than those with a smaller index. Typically, the number of objects increases as the index grows, but for example increasing the number of airplanes in the logistics problems actually makes the problems much easier. So, there is not always a direct connection between the index and the difficulty, as is often apparent from the runtimes.

1-linearization semantics is usually better than the other semantics, often one order of magnitude better, sometimes two. This always goes back to the shorter parallel plan length: formulae are smaller and therefore in general easier to evaluate. However, smaller formula size alone is not the complete explanation, and we should do more work on determining how strongly constrained these formulae are in

instance	len	val	1-lin	step	proc
depot-10-7654	7	F	0.01		
depot-10-7654	8	T	0.02		
depot-10-7654	9	F		0.28	0.30
depot-10-7654	10	T		0.29	0.34
depot-11-8765	13	F	0.04		
depot-11-8765	14	T	0.44		
depot-11-8765	17	F		69.56	70.29
depot-11-8765	18	?		-	-
depot-11-8765	19	?		-	-
depot-11-8765	20	T		$>$ 154.43	$>$ 157.28
depot-12-9876	19	F	0.24		
depot-12-9876	20	T	$>$ 143.73		
depot-12-9876	21	F		142.12	$>$ 140.75
depot-12-9876	22	?		-	-
depot-13-5646	7	F	0.01		
depot-13-5646	8	T	0.01		
depot-13-5646	8	F		0.01	0.01
depot-13-5646	9	T		0.04	0.05
depot-14-7654	9	F	0.05		
depot-14-7654	10	T	0.11		
depot-14-7654	11	F		1.25	1.28
depot-14-7654	12	T		2.97	2.89
depot-15-4534	17	F	0.15		
depot-15-4534	18	T	41.78		
depot-15-4534	19	F		$>$ 120.55	$>$ 117.82
depot-15-4534	20	?		-	-
depot-15-4534	21	?		-	-
depot-15-4534	22	?		-	-
depot-15-4534	23	T		$>$ 193.58	-
depot-16-4398	7	F	0.01		
depot-16-4398	8	T	0.01		
depot-16-4398	7	F		0.01	0.01
depot-16-4398	8	T		0.07	0.08
depot-17-6587	5	F	0.01		
depot-17-6587	6	T	0.01		
depot-17-6587	6	F		0.02	0.02
depot-17-6587	7	T		0.10	0.10
depot-18-1916	11	F	0.26		
depot-18-1916	12	T	4.95		
depot-18-1916	11	F		0.15	0.15
depot-18-1916	12	T		58.00	57.98
depot-19-6178	9	F	0.05		
depot-19-6178	10	T	0.05		
depot-19-6178	9	F		0.09	0.09
depot-19-6178	10	T		0.80	0.74

Table 1: Runtimes of Depot problems in seconds

instance	len	val	1-lin	step	proc
driver-2-3-6d	12	F	0.40		
driver-2-3-6d	13	T	0.64		
driver-2-3-6d	15	F		17.79	17.78
driver-2-3-6d	16	T		5.96	8.71
driver-2-3-6e	7	F	0.01		
driver-2-3-6e	8	T	0.03		
driver-2-3-6e	11	F		0.94	1.12
driver-2-3-6e	12	T		1.04	1.04
driver-3-3-6b	8	F	0.14		
driver-3-3-6b	9	T	0.09		
driver-3-3-6b	10	F		0.80	0.83
driver-3-3-6b	11	T		0.84	0.78
driver-4-4-8	8	F	0.12		
driver-4-4-8	9	T	0.12		
driver-4-4-8	10	F		1.27	1.29
driver-4-4-8	11	T		6.12	5.80
driver-5-5-10	13	F	37.61		
driver-5-5-10	14	?	-		
driver-5-5-10	15	T	$>$ 132.17		
driver-5-5-10	15	F		$>$ 124.34	128.73
driver-5-5-10	16	?		-	-
driver-5-5-10	17	?		-	-
driver-5-5-10	18	T		-	$>$ 178.07

Table 2: Runtimes of DriverLog problems in seconds

instance	len	val	1-lin	step	proc
log-20-0	8	F	0.22		
log-20-0	9	T	0.83		
log-20-0	14	F		9.39	9.38
log-20-0	15	T		49.53	38.59
log-20-1	8	F	0.53		
log-20-1	9	T	0.10		
log-20-1	14	F		49.73	25.80
log-20-1	15	?		-	-
log-20-1	16	T		22.49	26.75
log-21-0	8	F	0.27		
log-21-0	9	T	0.93		
log-21-0	14	F		32.79	15.08
log-21-0	15	?		-	-
log-21-0	16	T		$>$ 103.42	$>$ 76.14
log-21-1	7	F	0.02		
log-21-1	8	T	0.36		
log-21-1	13	F		5.39	5.65
log-21-1	14	T		21.81	17.42
log-22-0	8	F	0.33		
log-22-0	9	T	0.77		
log-22-0	15	F		$>$ 164.23	$>$ 156.26
log-22-0	16	T		58.17	39.53
log-22-1	8	F	22.25		
log-22-1	9	T	5.82		
log-22-1	13	F		$>$ 37.30	42.80
log-22-1	14	?		-	-
log-22-1	15	?		-	-
log-22-1	16	T		$>$ 119.27	$>$ 122.94

Table 3: Runtimes of Logistics problems in seconds

instance	len	val	1-lin	step	proc
satell-11	4	F	0.01		
satell-11	5	T	0.06		
satell-11	7	F		0.20	0.20
satell-11	8	T		0.18	0.24
satell-12	7	F	28.42		
satell-12	8	T	3.02		
satell-12	13	F		104.84	$>$ 88.59
satell-12	14	T		5.35	6.75
satell-13	6	F	9.03		
satell-13	7	T	6.43		
satell-13	12	F		30.58	38.54
satell-13	13	T		44.19	39.38
satell-14	4	F	9.53		
satell-14	5	T	1.90		
satell-14	7	F		27.39	22.52
satell-14	8	T		4.23	3.65
satell-15	4	F	8.86		
satell-15	5	T	1.65		
satell-15	7	F		24.15	21.91
satell-15	8	T		3.61	3.50
satell-16	3	F	2.27		
satell-16	4	T	3.91		
satell-16	5	F		9.17	8.24
satell-16	6	?		-	-
satell-16	7	T		6.57	6.85
satell-17	3	F	0.22		
satell-17	4	T	2.48		
satell-17	5	F		1.12	1.31
satell-17	6	T		1.86	1.96
satell-18	4	F	0.06		
satell-18	5	T	0.23		
satell-18	7	F		0.24	0.27
satell-18	8	T		0.48	0.57
satell-19	6	F	46.26		
satell-19	7	T	25.78		
satell-19	10	F		$>$ 225.50	-
satell-19	11	?		-	-
satell-19	12	T		$>$ 170.69	-

Table 4: Runtimes of Satellite problems in seconds

instance	len	val	1-lin	step	proc
zeno-3-10	4	F	0.05		
zeno-3-10	5	T	0.06		
zeno-3-10	6	F		0.54	0.54
zeno-3-10	7	T		0.61	0.50
zeno-5-10	3	F	0.10		
zeno-5-10	4	T	0.26		
zeno-5-10	5	F		1.54	1.42
zeno-5-10	6	T		2.76	2.59
zeno-5-15	5	F	$>$ 144.02		
zeno-5-15	6	T	23.70		
zeno-5-15	5	F		1.96	1.88
zeno-5-15	6	?		-	-
zeno-5-15	7	T		42.38	45.34

Table 5: Runtimes of Zeno problems in seconds

comparison to the formulae for step semantics in order to offer a full explanation of the efficiency gains.

Constraints derived from process semantics usually do not – surprisingly and contrary to our expectations – provide an advantage over step semantics although there are often many fewer potential plans to consider. In few cases, like for the last unsatisfiable formulae for log-20-1 and log-21-0, process constraints halve the runtimes. The reason for the ineffectiveness of process semantics may lie in the nature of these benchmarks: there are often many operators potentially preventing the earlier application of an operator, and the constraints start pruning the search space only very late in the search when all or almost all operators have been chosen not to be applied. We plan to extend our experiments to other types of problems to see how process semantics more generally behaves.

Related Work

Kautz and Selman (1996; 1999) developed the planning as satisfiability approach and were the first to use the step semantics. Later work has addressed different problem representations (Kautz & Selman 1996) and implemented the same ideas in different computational frameworks like integer programming (Vossen *et al.* 1999) and nonmonotonic logic programs (Dimopoulos, Nebel, & Koehler 1997).

The BLACKBOX planner of Kautz and Selman (1999) is the best-known planner implementing the satisfiability planning paradigm, and its encodings are close to those of ours for the step semantics. There is a bug in the currently distributed version of BLACKBOX and we were not able to make a broader comparison to BLACKBOX, but we will do this as soon as a working version of the planner is available. It seems that formulae produced by BLACKBOX are several times bigger than our formulae for step semantics, and solution times are of the same order of magnitude.

The process semantics discussed in this paper was first considered in connection with Petri nets; for an overview see (Best & Devillers 1987).

Conclusions

We have presented efficient translations of parallel planning to SAT, and shown that some of these semantics are very effective, often one or two orders of magnitude faster, than the standard semantics of parallelism used by the main algorithms utilizing the inherent parallelism in planning problems. The best semantics for many of these problems turned out to be the 1-linearization semantics, which provides the most compact problem encodings in terms of number of time steps and number of constraints related to parallelism. Interestingly, the process semantics, a refinement of the standard step semantics imposing a further condition on plans, did not significantly improve planning efficiency. One topic for future research would be to combine process semantics and 1-linearization semantics.

The constraints on parallel application of operators have in general a quadratic size, and on many problems these dominate the size of the formulae, presumably worsening planning efficiency. More compact encodings, preferably having linear size, should be investigated.

Acknowledgements

We thank Lawrence Ryan for assistance with the Siege solver.

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 Cleaveland, W. R., ed., *Tools and Algorithms for the Construction and Analysis of Systems, Proceedings of 5th International Conference, TACAS'99*, volume 1579 of *Lecture Notes in Computer Science*, 193–207. Springer-Verlag.

Blum, A. L., and Furst, M. L. 1997. Fast planning through planning graph analysis. *Artificial Intelligence* 90(1-2):281–300.

Cayrol, M.; Régnier, P.; and Vidal, V. 2001. Least commitment in Graphplan. *Artificial Intelligence* 130(1):85–118.

Dimopoulos, Y.; Nebel, B.; and Koehler, J. 1997. Encoding planning problems in nonmonotonic logic programs. In Steel, S., and Alami, R., eds., *Recent Advances in AI Planning. Fourth European Conference on Planning (ECP'97)*, number 1348 in Lecture Notes in Computer Science, 169–181. Springer-Verlag.

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

Kautz, H., and Selman, B. 1996. Pushing the envelope: planning, propositional logic, and stochastic search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference*, 1194–1201. Menlo Park, California: AAAI Press.

Kautz, H., and Selman, B. 1999. Unifying SAT-based and graph-based planning. In Dean, T., ed., *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, 318–325. Morgan Kaufmann Publishers.

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

Tarjan, R. E. 1972. Depth first search and linear graph algorithms. *SIAM Journal on Computing* 1(2):146–160.

Vossen, T.; Ball, M.; Lotem, A.; and Nau, D. 1999. On the use of integer programming models in AI planning. In Dean, T., ed., *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, volume I, 304–309. Morgan Kaufmann Publishers.