In *Proc. 12th Int'l Conf. on Artificial Intelligence (IJCAI-91)*, Sydney, Australia, 1991, pp. 268–273

Parallel Non-Binary Planning in Polynomial Time

Christer Bäckström
Dept. of Computer Science,
Linköping University
S-581 83 Linköping, Sweden
email: cba@ida.liu.se

Inger Klein
Dept. of Electrical Engineering,
Linköping University
S-581 83 Linköping, Sweden
email: inger@isy.liu.se

Abstract

This paper formally presents a class of planning problems which allows non-binary state variables and parallel execution of actions. The class is proven to be tractable, and we provide a sound and complete polynomial time algorithm for planning within this class. This result means that we are getting closer to tackling realistic planning problems in sequential control, where a restricted problem representation is often sufficient, but where the size of the problems make tractability an important issue.

1 Introduction

A large proportion of earlier papers about planning focus either on implementation of planners, or on representation problems, using logic or otherwise, and do not address computational issues at all.

Among earlier work on planning complexity, Chapman [1987] has designed an algorithm, called TWEAK, which captures the essentials of constraint-posting nonlinear planners. TWEAK is proven correct, but does not always terminate. Chapman has proven that the class of problems TWEAK is designed for is undecidable. Dean and Boddy [1988] have investigated some classes of temporal projection problems with propositional state variables. They report that practically all but some trivial classes are NP. It should be noted, however, that they assume a non-deterministic domain where events actually occur only if their pre-conditions are fulfilled. Korf [1987] presents some complexity results for traditional search based planning. He shows how the complexity can be reduced for problems where subgoals are serializable or independent, and he also shows how macro-operators and abstract actions can reduce complexity under certain assumptions.

The majority of papers on temporal logics discuss representation of problems, and results about complexity and computability are almost non-existent. An implementation of a restricted version of one temporal logic, ETL, is reported by Hansson [1990]. His decision procedure solves temporal projection in exponential time, but is not guaranteed to terminate for planning. Recent work by van Beek [1990] presents some complexity results for some temporal ordering problems in a point algebra and a simplified interval algebra. These results are relevant, but not immediately applicable, to planning.

Chapman [1987] says: 'The restrictions on action representation make TWEAK almost useless as a real-world planner.' He also says: 'Any Turing machine with its input can be encoded in the TWEAK representation.' It seems that any useful class of planning problems is necessarily undecidable. However, we think that a planner that is capable of encoding a Turing machine has much more power than needed for most problems. It seems that TWEAK is too limited in some aspects, but overly expressive in other aspects. We think that finding classes of problems that balance such aspects against each other, so that they are decidable or even tractable, is an important and interesting research challenge. On the other hand, we should probably not have much hope of finding one single general planner with such properties. The research task is rather to find different classes of problems which are strong in different aspects, so as to be tuned to different kinds of application problems.

We have focussed our research on problems where the action representation is even more restricted than in TWEAK, but where we can prove interesting theoretical properties. Our intended applications are in the area of sequential control, a subfield of control theory, where a restricted problem representation is often sufficient, but where the size of the problems make tractability an important issue (see section 6).

In previous papers [Bäckström and Klein, 1990a; Bäckström and Klein, 1990b] we have presented a polynomial-time, $O(m^3)$ in the number of state variables, planning algorithm for a limited class of planning problems, the SAS-PUBS class. Compared to previous work on complexity of algorithms for knowledge-based or logic-based planning, our algorithm achieves computational tractability, but at the expense of only applying to a significantly more limited class of problems. Our general research strategy is to start with a restricted but tractable class of planning problems and to then gradually extend this class while establishing its properties after each such step. This is a very usual strategy in most disciplines of science, but is, unfortunately, not very common in AI. A similar strategy has been pursued by Brachman and Levesque [1984] who have studied the relationship between generality and tractability in knowledge representation languages.

The SAS-PUBS class constitutes the first step along the strategy just outlined, but this class is probably too simple to be of other than theoretical interest. However, even with very moderate extensions one would probably obtain problem classes that occur frequently in practice. This paper presents the *SAS-PUS* class, which is an extension of the SAS-PUBS class and which brings us closer to reality. We prove this class also to be tractable by presenting a sound and complete polynomial time planning algorithm for it. Furthermore, the algorithm only orders those actions that must necessarily be executed in sequence and allows for parallel execution of unordered actions.

2 Ontology of Worlds, Actions and Plans

This section defines our planning ontology with the main concepts being: world states, actions, and plans. Although presented in a slightly different way, the ontology is essentially as described by Sandewall and Rönnquist [1986]. For further explanation and intuition regarding action structures, the reader is referred to Sandewall and Rönnquists paper.

2.1 World Description

We assume that the world can be modelled by a finite number of *features*, or state variables, where each feature can take on values from some finite discrete domain, or the value *undefined*, u. For technical reasons, the contradictory value, k, is added. The combination of the values of all features is a *partial state*, and if no values are undefined the state is also a *total state*. If it is clear from the context, or if it does not matter whether a state is total or not, we simply call it a *state*. The order $\sqsubseteq$, reflecting information content, is defined on the feature values s.t. the undefined and contradictory values contain less and more information, respectively, than all the other values. These other values contain equal amount of information and are mutually incomparable.

Definition 2.1

1. $\mathcal{M}$ is a finite set of *feature indices*.
2. $\mathcal{S}_i$, where $i \in \mathcal{M}$, is the *domain* for the i:th feature. $\mathcal{S}_i$ must be finite. $\mathcal{S}_i^+ = \mathcal{S}_i \cup \{u, k\}$ where $i \in \mathcal{M}$ is the *i:th extended domain*. $\mathcal{S} = \prod_{i \in \mathcal{M}} \mathcal{S}_i$ is the *total state space* and $\mathcal{S}^+ = \prod_{i \in \mathcal{M}} \mathcal{S}_i^+$ is the *partial state space*.
3. $s[i]$ for $s \in \mathcal{S}^+$ and $i \in \mathcal{M}$ denotes the value of the i:th feature of s. The function $dim : \mathcal{S}^+ \rightarrow 2^{\mathcal{M}}$ is defined s.t. for $s \in \mathcal{S}^+$, $dim(s)$ is the set of all feature indices i s.t. $s[i] \neq u$. If $i \in \text{dim}(s)$ then i is said to be *defined* for s. A state $s \in \mathcal{S}^+$ is said to be *consistent* if $s[i] \neq k$ for all $i \in \mathcal{M}$.
4. $\sqsubseteq$ is a reflexive partial order on $\mathcal{S}_i^+$ defined as
$$\forall x, x' \in \mathcal{S}_i^+ (x \sqsubseteq x' \leftrightarrow x = u \vee x = x' \vee x' = k)$$
$\langle \mathcal{S}_i^+, \sqsubseteq \rangle$ forms a flat lattice for each i.
5. $\sqsubseteq$ is a reflexive partial order over $\mathcal{S}^+$ defined as
$$\forall s, s' \in \mathcal{S}^+ (s \sqsubseteq s' \leftrightarrow \forall i \in \mathcal{M}(s[i] \sqsubseteq s'[i]))$$
so $\langle \mathcal{S}^+, \sqsubseteq \rangle$ forms a lattice. □

The lattice operations $\sqcup$, join, and $\sqcap$, meet, are defined as usual on the lattices $\langle \mathcal{S}_i^+, \sqsubseteq \rangle$ and $\langle \mathcal{S}^+, \sqsubseteq \rangle$.

2.2 Action Types and Actions

Plans are constituted by *actions*, the atomic objects that will have some effect on the world when the plan is executed. Each action in a plan is a unique *occurrence*, or instantiation, of an *action type*, the latter being the specification of how the action 'behaves'. Two actions are of the same type iff they behave in exactly the same way. The 'behaviour' of an action type is defined by three partial state valued functions, the *pre-*, the *post-*, and the *prevail-condition*. Given an action, the conditions of its type are interpreted as follows: the pre-condition states what must hold at the beginning of the action, the post-condition states what will hold at the end of the action, and the prevail-condition states what must hold during the action. One could think of the pre- and post-conditions as defining non-sharable resources and the prevail-condition as defining sharable resources, using operating systems terminology.

Every action type is subject to the following constraints: all conditions must be consistent, the pre- and post-conditions must define exactly the same features, the pre- and post-condition must not specify the same value for any of their defined features, and a feature defined in the pre-condition must not be defined in the prevail-condition. We also demand that two distinct action types must differ in at least one condition.

In order to distinguish actions of the same type we attach a unique *label* to each action. We also let an action 'inherit' the conditions from its associated action type.

Definition 2.2

1. $\mathcal{H}$ is a set of action types.
2. $b, e, f : \mathcal{H} \rightarrow \mathcal{S}^+$ are functions giving the pre-, post- and prevail-condition, respectively, of an action type.
3. $\mathcal{L}$ is an infinite set of *action labels*.
4. A set $\mathcal{A} \subseteq \mathcal{L} \times \mathcal{H}$ is a set of *actions* iff no two distinct elements in $\mathcal{A}$ have identical labels. □

2.3 Plans

An ordered set of actions is a *plan* from one total state to another total state iff, when starting in the first state, we end up in the second state after executing the actions of the plan in the specified order. The plan is *linear* if the set is totally ordered, and it is *non-linear* if it is partially ordered. In a non-linear plan, the order between two actions does not have to be specified if these actions can be executed in arbitrary order. The persistence handling essentially uses the STRIPS assumption [Fikes and Nilsson, 1971], and, since the formalism is very restricted, the frame problem [Hayes, 1981; Brown, 1987] is also avoided. The definition of plans is based on the relation $\longmapsto$ which defines how ordered sets of actions can transform one state into another.

Definition 2.3 The relation $\longmapsto \subseteq \mathcal{S} \times 2^{(\mathcal{L} \times \mathcal{H})} \times 2^{(\mathcal{L} \times \mathcal{H})^2} \times \mathcal{S}$ is defined s.t. if $s, s' \in \mathcal{S}$, Ψ is a set of actions, $a \in \Psi$, and σ is a total order on Ψ then $\longmapsto$ is defined as:

1. $s \overset{\varnothing,\varnothing}{\longmapsto} s$
2. $s \overset{\{a\},\varnothing}{\longmapsto} s'$ iff $b(a) \sqcup f(a) \sqsubseteq s$, $e(a) \sqcup f(a) \sqsubseteq s'$ and $s[i] = s'[i]$ for all $i \notin dim(b(a) \sqcup f(a))$
3. $s \overset{\Psi,\sigma}{\longmapsto} s'$ where $|\Psi| \geq 2$ iff $a_1, \ldots, a_n$ are the actions in Ψ in the order σ and there are states $s_1, \ldots, s_n \in \mathcal{S}$ s.t. $s = s_0$, $s' = s_n$ and $s_{k-1} \overset{\{a_k\},\varnothing}{\longmapsto} s_k$ for $1 \leq k \leq n$. □

Definition 2.4 A tuple $\langle \Psi, \rho \rangle$ is a *linear plan* from s_o to $s_\star$ iff Ψ is a set of actions and ρ is a total order on Ψ s.t. $s_o \overset{\Psi,\rho}{\longmapsto} s_\star$. Similarly, a tuple $\langle \Psi, \rho \rangle$ where ρ is a partial order on Ψ is a *non-linear plan* from s_o to $s_\star$ iff $\langle \Psi, \sigma \rangle$ is a linear plan for any total order σ on Ψ s.t. $\rho \subseteq \sigma$. □

A plan for a specific problem is *minimal* iff there is no other plan solving the same problem using fewer actions.

Definition 2.5 A plan $\langle \Psi, \rho \rangle$ from s_o to $s_\star$ is *minimal* iff there is no other plan $\langle \Phi, \sigma \rangle$ from s_o to $s_\star$ s.t. $|\Phi| < |\Psi|$. □

We say that two actions are *independent*, meaning they can be executed in parallel, iff any feature changed by one of the actions is undefined in all conditions of the other action and whenever both actions define the same feature in their prevail-conditions, they define the same value for this feature. We further say that a plan is a *parallel plan* if all its unordered actions are independent, and it is *maximally parallel* if no pair of independent actions is ordered.

Definition 2.6 Two actions a and a' are *independent* iff, for all $i \in \mathcal{M}$, all of the following conditions hold:

1. $b(a)[i] \neq u$ implies $b(a')[i] \sqcup f(a')[i] = u$
2. $b(a')[i] \neq u$ implies $b(a)[i] \sqcup f(a)[i] = u$
3. $f(a)[i] \sqsubseteq f(a')[i]$ or $f(a')[i] \sqsubseteq f(a)[i]$ □

Definition 2.7 A non-linear plan $\langle \Psi, \rho \rangle$ from s_o to $s_\star$ is a *parallel* plan iff all pairs of actions $a, a' \in \Psi$ s.t. neither $a \rho a'$ nor $a' \rho a$ are independent. □

Definition 2.8 A parallel plan $\langle \Psi, \rho \rangle$ is *maximally parallel* iff $\langle \Psi, \sigma \rangle$ is not parallel for any $\sigma \subset \rho$. □

3 Classes of Planning Problems

The class of planning problems according to our ontology so far is called the *SAS*, Simplified Action Structures, class.

We also want to talk about more restricted classes, so we define some useful properties that can be ascribed to problem classes. We say that a domain is *binary* if it has exactly two elements, $|\mathcal{S}_i| = 2$. The set of action types is *unary* if all action types change exactly one feature, *post-unique* if no two different action types can change a certain feature to the same value, and *single-valued* if no two action types have defined but different values of their prevail-conditions for the same feature.

The rest of this paper concentrates on the *SAS-PUS* class (PUS meaning post-unique, unary, and single-valued). The implications of these restriction are discussed in section 6. The SAS-PUS class is an extension of the previously presented SAS-PUBS class, which also requires the domains to be binary.

Definition 3.1 A planning problem is in the SAS-PUS class iff it is SAS and $\mathcal{H}$ is unary, post-unique and single-valued. □

4 SAS-PUS Planning

This section presents an algorithm for finding minimal plans for the SAS-PUS class and also states some theoretical results about the algorithm and the SAS-PUS class. The proofs are omitted because of the page limit, but they can be found in our report [Bäckström and Klein, 1991].

The definitions of most functions and procedures used in the algorithm should be obvious, but the following three might need some explanation.

FindActionPost(A,i,x) Searches the set A of actions for a member a s.t. $e(a)[i] = x[i]$ and which is returned if it exists. If such an a does not exist, the value **nil** is returned.

FindAndRemove(A,i,x) Like *FindActionPost* but also removes a from A.

Order(a,a',r) Adds ara' to the relation r.

Algorithm 4.1

Input: $\mathcal{M}$, a set of feature indices, $\mathcal{A}$, a set containing two actions of each type in $\mathcal{H}$, and s_o and $s_\star$, the initial and final states respectively.

Output: D, a set of actions, and r a relation on D.

```
Procedure Plan($s_o, s_\star$ : state; $\mathcal{M}$ : set of
  feature indices; $\mathcal{A}$ : set of actions);
 var $i$ : feature index; $a, a'$ : action;
  $D, P, T$ : set of actions; $L$ : list of actions;
  $r$ : relation on $D$;

Procedure BuildChain($s_F, s_T$ : state; )
  $i$ : featureindex; $A$ : set of actions; $D, T$ : in out
  set of actions; $r$ : in out relation);
vars : state; $a, a'$ : action; $L$ : list of actions;
begin{BuildChain}
  $L$ := nil; $a'$ := nil; $s := s_T$;
  while $s[i] \neq s_F[i]$ do
    $a$ := FindAndRemove($A, i, s$);
    if $a$ = nil then fail
    else
      Insert($a, D$); Insert($a, T$); Concat($a, L$);
      if $a' \neq$ nil then Order($a, a', r$)
      $a' := a$; $s := b(a)$;
  return $L$;
end; {BuildChain}

begin{Plan}
  $D := \varnothing$; $T := \varnothing$; $r := \varnothing$;
  for $i \in \mathcal{M}$ do
```

```
        $L := BuildChain(s_o, s_\star, i, \mathcal{A}, D, T, r)$;
      $P := Copy(D)$;
      **while** $T \neq \emptyset$ **do**
        $a := RemoveAnAction(T)$;
        **for** $i \in \mathcal{M}$ **do**
          **if** $f(a)[i] \not\sqsubseteq s_o[i]$ **then**
            $a' := FindActionPost(D, i, f(a))$;
            **if** $a' \neq$ **nil then** $Order(a', a, r)$
            **else**
              $L := BuildChain(s_o, f(a), i, \mathcal{A}, D, T, r)$;
              $Order(Last(L), a, r)$;
          **if** $f(a)[i] \not\sqsubseteq s_\star[i]$ **then**
            $a' := FindActionPre(D, i, f(a))$;
            **if** $a' \neq$ **nil then** $Order(a, a', r)$
            **else**
              $L := BuildChain(f(a), s_o, i, \mathcal{A}, D, T, r)$;
              $Order(a, First(L), r)$;
              $a' := FindActionPre(P, i, s_o)$;
              **if** $a' \neq$ **nil then** $Order(Last(L), a', r)$
      $TransitiveClosure(r)$;
      **if** *"r is not antireflexive"* **then fail**
      **return** $\langle D, r\rangle$;
    **end**; {Plan}
```

The main variables are D, T and r. D is a non-decreasing set of actions which will eventually be the set of actions in the plan, if the algorithm succeeds. Every action ever inserted into D is also inserted into T, and the use of this set will become clear later on. r is a relation on D, and it will eventually be the execution order of the plan.

The function *BuildChain* has the purpose of trying to find a, possibly empty, sequence of actions in A which, if executable, changes the i:th feature from $s_F[i]$ to $s_T[i]$. If such a sequence is found, it is removed from A and inserted into D and T. Otherwise, the algorithm fails.

The main body of the algorithm first calls *BuildChain* once for each feature i to find a sequence of action changing i from $s_o[i]$ to $s_\star[i]$. D now contains all actions primarily needed to change s_o into $s_\star$, but all of these actions do not necessarily have their prevail-conditions satisfied. The purpose of the while loop in the algorithm is to achieve that all actions have their prevail-conditions satisfied. Since all actions in T are eventually removed from T and processed by the body of the while loop, all actions in the final plan will have their prevail-conditions satisfied. For each action a in T, the body of the while loop tests, for each feature i, whether the prevail-condition of the current action is satisfied in s_o. Nothing need be done if this is the case, but, otherwise, the algorithm checks if there is already a sequence of actions in D that changes the i:th feature from s_o to $f(a)$. If there is such a sequence, it is ordered before a, and, otherwise, *BuildChain* is called to find such a sequence. Since the actions needed to satisfy the prevail-condition of a might interfere with the primary actions changing s_o into $s_\star$, we must also assure that $f(a)[i]$ is changed into $s_\star[i]$. This is done in the second half of the body of the while loop, and in a way analogous to the first part. The difference is that if *BuildChain* is called, it finds a sequence of actions changing $f(a)[i]$ into $s_o[i]$, not $s_\star[i]$. The reason for this is that if $s_o[i] \neq s_\star[i]$, then there is already a sequence of actions in D changing $s_o[i]$ into $s_\star[i]$, and which is then ordered after the newly found sequence. After computing the transitive closure of r, it is tested for antireflexivity, and the algorithm fails if the order contains circularities. The algorithm is proven sound and complete.

Theorem 4.1 Given a SAS-PUS planning problem, if there is any plan solving the problem then algorithm 4.1 finds a minimal non-linear plan that solves the problem, otherwise it fails. $\square$

The main reason that the algorithm can be so simple is that the set of action types is single-valued. This gives as a result that any action affecting a certain feature is either ordered before or after all actions defining this feature in their prevail-conditions. In other words, all actions defining a certain feature in their prevail-conditions can share action sequences achieving this value and assuring the final value. This is also the reason that no plan contains more than two actions of each type.

It can also be proven that the plan returned by the algorithm is maximally parallel.

Theorem 4.2 Algorithm 4.1 can be implemented to run in $O(m^3n^3)$ time using $O(m^2n^2)$ space where $m = |\mathcal{M}|$ and $n = \max_{i\in\mathcal{M}} |\mathcal{S}_i|$. $\square$

Both complexity figures can be reduced by more detailed analysis of the size of $\mathcal{H}$, the difference in domain sizes, and other factors. It should also be noted that the only data structure requiring more than $O(mn)$ space is the output data, and that we are not likely to be interested in the transitive closure in practical applications.

The main explanation for the complexity result is that the set of action types is post-unique and single-valued. Post-uniqueness implies that there is never any choice of which action type to use. Single-valuedness implies, as was mentioned above, that no plan contains more than two actions of each type, which thus bounds the number of iterations of the main while loop.

5 Example

This section presents an example that fits in the SAS-PUS class. The example is a much simplified version of a LEGO[1] car factory which is used for undergraduate laborations in sequential control at Linköping University [Strömberg, 1990]. The task is to assemble a LEGO car from pre-assembled parts as shown in figure 1.

We represent the problem using three features defined as follows:

$s[1]$: 1: Chassis in chassis storage, 2: Chassis at workstation

$s[2]$: 1: Top in top storage, 2: Top at workstation, 3: Top on chassis

$s[3]$: 1: Wheels in wheel storage, 2: Wheels at workstation, 3: Wheels on chassis

Obviously, $\mathcal{M} = \{1, 2, 3\}$ and states are written as $\langle s[1], s[2], s[3]\rangle$. We assume that the set $\mathcal{H}$ consists of the

[1]LEGO is a trademark of the LEGO Group.

h	b(h)	e(h)	f(h)	Explanation
h_1	$\langle 1,u,u\rangle$	$\langle 2,u,u\rangle$	$\langle u,u,u\rangle$	Move chassis to workstation
h_2	$\langle 2,u,u\rangle$	$\langle 1,u,u\rangle$	$\langle u,u,u\rangle$	Move chassis to chassis storage
h_3	$\langle u,1,u\rangle$	$\langle u,2,u\rangle$	$\langle u,u,u\rangle$	Move top to work station
h_4	$\langle u,2,u\rangle$	$\langle u,3,u\rangle$	$\langle 2,u,u\rangle$	Mount top
h_5	$\langle u,u,1\rangle$	$\langle u,u,2\rangle$	$\langle u,u,u\rangle$	Move wheels to work station
h_6	$\langle u,u,2\rangle$	$\langle u,u,3\rangle$	$\langle 2,u,u\rangle$	Mount wheels

Table 1: Action types for the example

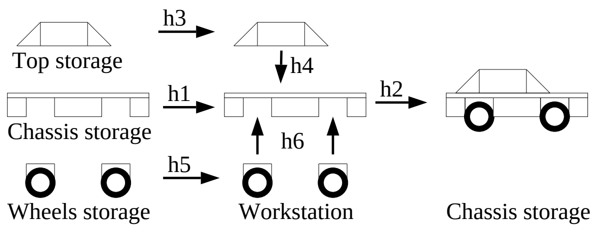

Figure 1: The LEGO car example

action types in table 1, and that the set $\mathcal{A}$ is $\{a_1,\ldots,a_{12}\}$ where a_i and a_{i+6} are of type h_i for $1 \leq i \leq 6$.

We also assume that the initial state is $s_o = \langle 1,1,1\rangle$, all parts in storage, and the final state is $s_\star = \langle 1,3,3\rangle$, an assembled car in the chassis storage.

The algorithm first calls *BuildChain* once for each $i \in \mathcal{M}$ to change $s_o[i]$ into $s_\star[i]$. *BuildChain* finds the empty sequence for feature 1, and the sequences a_3, a_4 and a_5, a_6 for features 2 and 3, respectively. Now, $\mathcal{A} = \{a_1, a_2, a_7, \ldots, a_{12}\}$, $D = T = \{a_3, a_4, a_5, a_6\}$ and r consists of a_2ra_4 and a_5ra_6.

The actions in T, plus those added to T during the following process, are removed one at a time and processed by the while loop. a_3 and a_5 fall straight through the loop body since their prevail-conditions are trivially satisfied. The prevail-condition of a_4 is not satisfied in the initial state, and there are no actions in D providing the prevail-condition of a_4. *BuildChain* is thus called, and it returns the action a_1 which changes the 1st feature to satisfy the prevail-condition of a_4. Similarly, *BuildChain* finds the action a_2 which assures the desired final value of this feature. Both these actions are inserted into D and T, and a_1ra_4 and a_4ra_2 are inserted into r. The action a_6 has the same prevail-condition as a_4, so there are already actions, namely a_1 and a_2, in D that provide its prevail-condition and assures the final value of the 1st feature. No actions are inserted into D, but r is extended with a_1ra_6 and a_6ra_2. The actions a_1 and a_2 are also removed from T, but their prevail-conditions are trivially satisfied.

The algorithm then computes the transitive closure of r, and, since r is anti-reflexive, it succeeds and returns the plan $\langle D, r\rangle$, where D is $\{a_1,\ldots,a_6\}$ and r is as depicted in figure 2.

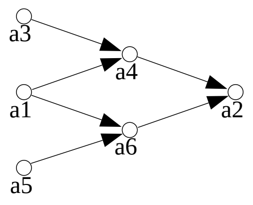

Figure 2: The plan for the example (transitive arcs omitted).

6 Discussion

The restriction that $\mathcal{H}$ be unary is serious for planning problems where two or more features can change simultaneously, but it is not always the same combinations of features that change simultaneously. Allowing non-binary domains does not help much in this case. Although one could represent several feature domains as one multi-valued feature, this would most likely violate the restrictions on action types in the SAS class. Post-uniqueness need not be a very limiting restriction for applications where there is little or no choice what plan to use, and where the size of the problem is the main difficulty when planning. However, for problems where $\mathcal{H}$ is non-unary or not single-valued, the major problem can be to choose between several different ways of achieving the goal. In this case, it will usually be impossible to make a post-unique formalization of the problem. The most serious restriction for the majority of practical applications is, in our opinion, the restriction that $\mathcal{H}$ is single-valued. As an example, requiring single-valuedness prevents us from modelling a problem where one action type requires a certain valve to be open and some other action type requires the same valve to be closed in their prevail-conditions.

The class one gets when relaxing the single-valuedness restriction is likely to be very interesting from a practical point of view. This class is conjectured sufficient for representing some interesting classes of real-world problems in *sequential control*, a subfield of *discrete event systems* within control theory. Examples of application areas are process plants and automated manufacturing. A particularly interesting problem here is to restart a process after a break-down or an emergency stop. After such an event, the process may be in anyone of a very large number of states, and it is not realistic to have precompiled plans for how to get the process back to normal again from any such state. Restarting is usually done manually and often by trial-and-error, and it is thus an application where automated planning is very relevant. It is interesting to note that such plans are complex because of their size, not because of complex actions. A process plant like a paper mill can have tens of thousands of sensors and actuators, so the number of features can be very large. It is easy to realize that the complexity issues are very important in this kind of applications.

Since single-valuedness seems to be the most serious restriction, it would be natural to try to eliminate that restriction first. Unfortunately, it can be shown that the resulting class of problems is intractable. However, this is because the plans themselves are of exponential size in the worst case, and such plans are unlikely to be of practical interest. We believe that it is possible to replace single-valuedness with other restrictions that are fulfilled for many practical problems, but which reduces the complexity drastically, and we are currently investigating such restrictions.

It would also be interesting to try to combine results along the line in this paper with the work on extended actions structures, allowing interdependent parallel actions or interval-valued features [Bäckström, 1988a; Bäckström, 1988b; Bäckström, 1988c].

7 Conclusion

We have identified a class of deterministic planning problems, the SAS-PUS class, which allows non-binary state variables and parallel actions. We have also presented a sound and complete polynomial time algorithm for finding minimal plans in this class. This result provides a kind of lower bound for planning; at least this class of problems is tractable. Since the SAS-PUS class is an extension of the previously presented SAS-PUBS class, we have managed to take a step upwards in expressibility while retaining tractability.

References

[Bäckström, 1988a] Christer Bäckström. Action structures with implicit coordination. In *Proceedings of the Third International Conference on Artificial Intelligence: Methodology, Systems, Applications (AIMSA-88)*, pages 103–110, Varna, Bulgaria, September 1988. North-Holland.

[Bäckström, 1988b] Christer Bäckström. Reasoning about interdependent actions. Licentiate Thesis 139, Department of Computer and Information Science, Linköping University, Linköping, Sweden, June 1988.

[Bäckström, 1988c] Christer Bäckström. A representation of coordinated actions characterized by interval valued conditions. In *Proceedings of the Third International Symposium on Methodologies for Intelligent systems (ISMIS-88)*, pages 220–229, Torino, Italy, October 1988. North-Holland.

[Bäckström and Klein, 1990a] Christer Bäckström and Inger Klein. Planning in polynomial time: The SAS-PUBS class. Research Report LiTH-IDA-R-90-16, Department of Computer and Information Science, Linköping University, Linköping, Sweden, August 1990.

[Bäckström and Klein, 1990b] Christer Bäckström and Inger Klein. Planning in polynomial time. In *Expert Systems in Engineering: Principles and Applications. International Workshop.*, pages 103–118, Vienna, Austria, September 1990. Springer.

[Bäckström and Klein, 1991] Christer Bäckström and Inger Klein. Parallel non-binary planning in polynomial time: The SAS-PUS class. Research Report LiTH-IDA-R-91-11, Department of Computer and Information Science, Linköping University, Linköping, Sweden, April 1991.

[van Beek, 1990] Peter van Beek. Reasoning about qualitative temporal information. In *Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI-90)*, pages 728–734, Boston, Massachussettes, August 1990. MIT Press.

[Brachman and Levesque, 1984] Ronald J Brachman and Hector J Levesque. The tractability of subsumption in frame-based description languages. In *Proceedings of the Fourth National Conference on Artificial Intelligence (AAAI-84)*, pages 34–37, Austin, Texas, 1984.

[Brown, 1987] Frank Brown, editor. *The Frame Problem in Artificial Intelligence, Proceedings of the 1987 Workshop*, Lawrence, Kansas, April 1987. Morgan Kaufman.

[Chapman, 1987] David Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–377, 1987.

[Dean and Boddy, 1988] Thomas Dean and Mark Boddy. Reasoning about partially ordered events. *Artificial Intelligence*, 36:375–399, 1988.

[Fikes and Nilsson, 1971] Richard E Fikes and Nils J Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[Hansson, 1990] Christer Hansson. A prototype system for logical reasoning about time and action. Licentiate Thesis 203, Department of Computer and Information Science, Linköping University, Linköping, Sweden, January 1990.

[Hayes, 1981] Patrick J Hayes. The frame problem and related problems in artificial intelligence. In Bonnie Lynn Webber and Nils J. Nilsson, editors, *Readings in Artificial Intelligence*, pages 223–230. Morgan Kaufman, 1981.

[Korf, 1987] Richard E Korf. Planning as search: A quantitative approach. *Artificial Intelligence*, 33:65–88, 1987.

[Sandewall and Rönnquist, 1986] Erik Sandewall and Ralph Rönnquist. A representation of action structures. In *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI-86)*, pages 89–97, Philadelphia, Pennsylvania, August 1986. Morgan Kaufman.

[Strömberg, 1990] Jan-Erik Strömberg. Styrning av LEGO-bilfabrik. 2nd revised edition. Department of Electrical Engineering, Linköping University, February 1990.