Planning in Polynomial Time: The SAS-PUBS Class

Christer Bäckström
Dept. of Computer Science,
Linköping University
S-581 83 Linköping,
Sweden
Phone: +46 13282429
email: cba@ida.liu.se

Inger Klein
Dept. of Electrical Engineering,
Linköping University
S-581 83 Linköping,
Sweden
Phone: +46 13281665
email: inger@isy.liu.se

This article appears in *Computational Intelligence,* 7(3):181–197, Aug. 1991.

Abstract

This article describes a polynomial-time, $O(n^3)$, planning algorithm for a limited class of planning problems. 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 algorithm is proven correct, and it always returns a parallel minimal plan if there is a plan at all.

Keywords: Planning, Knowledge Representation, Complexity

1 Introduction

Almost all previous papers about planning and temporal reasoning has focussed either on implementations of planners or on theoretical aspects of the representation of time and actions. The bulk of papers in the first of these groups reflect the evolution of 'classical' constraint-posting planners from STRIPS (Fikes and Nilsson, 1971) to SIPE (Wilkins, 1988),the latter of these usually being considered as state of the art in planning. Unfortunately, these papers do not provide any deep theoretical analyses of the planning algorithms employed so not much is known about correctness and complexity of these algorithms. Wilkins (1988), for example, admits that it is hardly possible to carry out such an analysis of SIPE. He gives some arguments about complexity behaviour in test applications but there is no formal analysis and the figures presented are probably not worst-case figures. The second group consists mainly of papers on various temporal logics where those by Allen (1981; 1984) and Shoham (1987) are among the most prominent within AI. These papers do, however, usually not address computational aspects at all but only representational issues. We will briefly discuss some of the few papers that have tried to bridge this gap between theory and practice.

Chapman (1987) has designed a planning algorithm, TWEAK, that captures the essentials of most constraint-posting non-linear planners like STRIPS and SIPE while being clean enough to allow theoretical analysis. TWEAK is proven correct and complete, 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 make the somewhat strange assumption that an action occurs successfully if its preconditions are satisfied and otherwise does not occur at all. Since all actions are processed in this way independently of whether the preceding actions have occurred or not, it is not obvious that this result is applicable to many real problems.

The majority of papers on temporal logics discuss representation of problems, and results about complexity and computability are almost non-existent. Temporal predicate logics are usually more expressive than FOP, so we could hardly hope for decidability without hard restrictions on the expressibility. Classical propositional logics are, however, decidable, so there might be some hope for restricted propositional temporal logics. Unfortunately, most temporal logics use some kind of non-monotonic reasoning to reason about change, thus making them undecidable. An implementation of a restricted version of one such logic, ETL (Sandewall, 1988b; Sandewall, 1988c; Sandewall, 1988d; Sandewall, 1989), is reported by Hansson (1990). His decision procedure solves temporal projection in exponential time, but is not guaranteed to terminate for planning.

All these results seem very disappointing, but how bad is the situation really? Chapman (1987) says: 'The restrictions on action representation make TWEAK almost useless as a real-world planner.' However, he also says: 'Any Turing machine with its input can be encoded in the TWEAK representation.' It might seem as if any useful class of planning problems is necessarily undecidable. Our opinion, however, is that a planner that is capable of encoding a Turing machine is much more expressive than most problems require. 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, thus being 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* and *discrete event systems* within control theory, where a restricted problem representation is often sufficient but where the size of the problems make tractability an important issue. It is naturally desirable to be able to analyse discrete control systems as rigorously as can be done for ordinary continous systems. Obviously, correctness can be a very important issue here and real-time requirements rise the question of complexity. More on this issue can be found in section 7.

Our general research strategy is to first find 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, and it is in distinction to the paradigm 'tackle a hard problem and fail' which is too often employed in AI. A similar strategy has been advocated by Brachman and Levesque (1984; 1985) who have studied the trade-off between expressibility and tractability in knowledge representation languages.

This article presents our first step of this strategy. We have identified a class of planning problems, the *SAS-PUBS class*, for which we have devised a planner which finds parallel minimal plans. The planner is proven sound and complete and runs in polynomial time, $O(n^3)$, in the number of state variables. 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. The algorithm employs a strategy that slightly resembles the means-ends analysis used in GPS

(Newell and Simon, 1972). It first finds the actions necessary to transform the initial state into the goal state. These action are not necessarily executable, so the algorithm then finds the extra actions necessary to transform the initial state into a state where these first actions are executable plus actions to redo these effects again. However, these new action are not necessarily executable either, so this process is repeated until all actions in the plan are executable or we know that there is no plan at all. This latter condition is actually very simple and is implicit in the algorithm and it is thus implicitly proven correct since the algorithm is proven correct. This description of the algorithm is somewhat simplified since it does not work on complete states but rather on partial states treating each state variable separately. There is thus no search over the state space involved but the algorithm rather works in a kind of parallel way on subgoals expressed by partial states. Unfortunately, the SAS-PUBS class is probably too simple to be of other than theoretical interest. However, even very moderate extensions to this class would probably be sufficient to tackle a lot of problem classes that occur frequently in practice, for example in process control. A discussion of the restrictions of the SAS-PUBS class can be found in section 6.

Although this article is very theoretical with many pages of definitions, theorems and proofs, it should be possible for a reader to get the main ideas by reading only the english text and skipping the formal parts.

2 Ontology of Worlds, Actions and Plans

This section defines our planning ontology with the main concepts being: (world) states, actions and plans. The *world* is understood as the abstraction of the real world that we use for planning. Although presented in a slightly different way, the ontology is essentially *action structures* as described by Sandewall and Rönnquist (1986a; 1986b). The major difference is that we do not use explicit time points but order the actions themselves instead. We can still express that actions are allowed to occur in parallel but we cannot say for example that an action starts during the occurrence of another action. Because of this difference, we call our ontology *simplified action structures* or simply *SAS*.

The reason we use action structures instead of more traditional planning formalism is that action structures imposes more structure on actions. Although limiting expressivity, this extra structure is advantageous for computational reasons. One could, at least when considering only sequential plans, reformulate our ontology in a more traditional notation. However, this would lead to considerably more awkward and unclear definitions and proofs so we do not find this a good idea even if many readers would feel more comfortable with such a notation. We have also tried to keep our notation close the one used by Sandewall and Rönnquist in order not to introduce yet another new notation.

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 values u and k. The value u means *undefined* and should be interpreted as 'don't care' while the value k, *contradictory*, is introduced for technical reasons, that is, to get a lattice. The combination of the values of all features is called a *partial state*, and if no values are undefined the state is also called a *total state*, that is, a total state is also a partial 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*. An order, $\sqsubseteq$, reflecting information content, is defined on the feature values such that the undefined value contains less information than all other values, the contradictory value contains more information than all other values and the defined values contain equal amount of information and are mutually incomparable. The order $\sqsubseteq$ is also straightforwardly extended to states.

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.
   $\mathcal{S}_i^+ = \mathcal{S}_i \cup \{u_i, k_i\}$ where $i \in \mathcal{M}$ is the *extended domain* for the i:th feature.
   $\mathcal{S} = \prod_{i \in \mathcal{M}} \mathcal{S}_i$ is the *total state space*.
   $\mathcal{S}^+ = \prod_{i \in \mathcal{M}} \mathcal{S}_i^+$ 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 and is called the *projection of s onto i*. A state $s \in \mathcal{S}^+$ is said to be *consistent* if $s[i] \neq k_i$ for all $i \in \mathcal{M}$.
4. The function $dim : \mathcal{S}^+ \to 2^{\mathcal{M}}$ is defined s.t. for $s \in \mathcal{S}^+$, $dim(s)$ is the set of all feature indices i s.t. $s[i] \neq u_i$. If $i \in \text{dim}(s)$ then i is said to be *defined* for s.
5. $\sqsubseteq_i$ is a reflexive partial order[1] on $\mathcal{S}_i^+$ defined as

$$\forall x, x' \in \mathcal{S}_i^+ (x \sqsubseteq_i x' \leftrightarrow x = u_i \lor x = x' \lor x' = k_i)$$

[1]By *partial order* we understand a relation that is antireflexive and transitive, and by *reflexive partial order* we understand a relation that is reflexive, antisymmetric and transitive. The terminology for partial orders is very confused in the mathematical literature, but this definition is practical for our purposes and it agrees with Mendelson's (1987) definition.

$\langle \mathcal{S}_i^+, \sqsubseteq_i \rangle$ forms a flat lattice for each i.

6. $\sqsubseteq$ is a reflexive partial order over $\mathcal{S}^+$ defined as

$$\forall s, s' \in \mathcal{S}^+ (s \sqsubseteq s' \leftrightarrow \forall i \in \mathcal{M}(s[i] \sqsubseteq_i s'[i]))$$

□

Both $\langle \mathcal{S}^+, \sqsubseteq \rangle$ and all $\langle \mathcal{S}_i, \sqsubseteq_i \rangle$ form lattices so the operations $\sqcup$ (join) and $\sqcap$ (meet) are defined in the usual way.

We will henceforth drop the subscripts of u_i, k_i and $\sqsubseteq_i$ and simply write u, k and $\sqsubseteq$ since no confusion is likely to occur.

Example 2.1 Let $\mathcal{M} = \{1,2\}$ and $\mathcal{S}_1 = \mathcal{S}_2 = \{0,1\}$, then $\mathcal{S}_1^+ = \mathcal{S}_2^+ = \{0,1,u,k\}$, $\mathcal{S} = \mathcal{S}_1 \times \mathcal{S}_2$ and $\mathcal{S}^+ = \mathcal{S}_1^+ \times \mathcal{S}_2^+$. The lattices $\langle \mathcal{S}_1^+, \sqsubseteq \rangle = \langle \mathcal{S}_2^+, \sqsubseteq \rangle$ and $\langle \mathcal{S}^+, \sqsubseteq \rangle$ are shown in figures 1 and 2.

Further suppose that we have three states $s_1 = \langle u, 1 \rangle$, $s_2 = \langle 0, 1 \rangle$ and $s_3 = \langle 1, k \rangle$, all in $\mathcal{S}^+$. For these states:

1. $s_1[1] = u$ and $s_1[2] = 1$.
2. Only states s_1 and s_2 are consistent.
3. $dim(s_1) = \{2\}$ and $dim(s_2) = dim(s_3) = \{1, 2\}$

□

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'. Actions and action types can be thought of as the steps and step templates respectively in TWEAK (Chapman, 1987). Two actions are of the same type iff they behave in exactly the same way. The 'behaviour definition' of an action type is defined by three partial state valued functions, the *pre-, post-* and *prevail-condition*. Given an action, the conditions of its corresponding type are interpreted as follows: the pre-condition states what must hold at the beginning of the action, the post-condition what will hold at the end of the action and the prevail-condition states what must hold during the action. The intuition behind these conditions is that the pre- and post-conditions express what effect the action has upon the world, that is, what feature(s) of the world it changes. The prevail-condition expresses which features must be constant during the execution and what the values must be for these features. An action changing a certain feature cannot be concurrent with another action also changing that same feature or specifying it to be constant in its prevail-condition. However, two actions defining the same feature in their prevail-conditions can be concurrent if they specify the same value for this feature. Making an analogy with operating systems theory, pre- and post-conditions can be thought of as expressing non-sharable resources and prevail-conditions as expressing sharable resources. There is, however, no such clear correspondence with the resources in SIPE (Wilkins, 1988). The consumable resources in SIPE could, at least to some extent, be represented with the pre- and post-conditions. Non-consumable resources can, on the other hand, not be represented in the SAS-formalism as described in this article, but they can be handled in action structures with keep-conditions (Bäckström, 1988a; Bäckström, 1988b).

If we were not considering parallel plans, the prevail-conditions would not be strictly necessary; a feature that is required for the execution of an action but not changed by it could be expressed either as defined in the pre-condition and undefined in the post-condition or defined with the same value in both pre- and post-condition. In fact, if taking the latter of these two approaches prevail-conditions would not even be necessary for parallel plans, but we find the theory much cleaner and clearer if such conditions are separated out as prevail-conditions so these are rather an asset than a burden.

Definition 2.2

1. $\mathcal{H}$ is a set of action types.
2. $b : \mathcal{H} \rightarrow \mathcal{S}^+$ gives the pre-condition of an action type
3. $e : \mathcal{H} \rightarrow \mathcal{S}^+$ gives the post-condition of an action type.
4. $f : \mathcal{H} \rightarrow \mathcal{S}^+$ gives the prevail-condition of an action type.

□

We further require our set $\mathcal{H}$ of action types to conform with the following axioms:

Axiom 2.3 $\forall h \in \mathcal{H} \forall i \in \mathcal{M}(b(h)[i] \neq k \wedge e(h)[i] \neq k \wedge f(h)[i] \neq k)$

Axiom 2.4 $\forall h \in \mathcal{H}(dim(b(h)) = dim(e(h)))$

Axiom 2.5 $\forall h \in \mathcal{H} \forall i \in dim(b(h))(b(h)[i] \neq e(h)[i])$

Axiom 2.6 $\forall h \in \mathcal{H}(dim(b(h)) \cap dim(f(h)) = \varnothing)$

Axiom 2.7 $\forall h, h' \in \mathcal{H}(b(h) = b(h') \land e(h) = e(h') \land f(h) = f(h') \to h = h')$

Axiom 2.3 expresses that all features must be consistent for all conditions of an action type. This is because the value k was introduced to make the domains form lattices and it is not really used. Axiom 2.4 requires all features defined in the pre-condition to be defined also in the post-condition and vice versa. This is admittedly a restriction since there might be applications where one wants to model actions that sets a feature to a certain value independently of its initial value. One might also want to model actions that require a feature to have a certain value when it starts executing but which leaves that feature undefined upon termination. It is out of the scope of the current article to investigate such extensions, but Sandewall (1988a) has paid some attention to this matter. Axiom 2.5 says that a feature that is defined in the pre-condition must have a different value in the post-condition. This is no real restriction since a feature that is defined but not changed by the action should be defined in the prevail-condition. The only problem could be if we want to model actions that require a feature to have a certain value when it starts executing and leaves that feature at the same value upon termination but where this feature is affected by the action during its execution and thus undefined there. Obviously this problem disappears if we restrict ourselves to sequential plans. It is far outside the scope of this article to deal with such a detailed representation of actions, but action structures with keep-conditions (Bäckström, 1988a; Bäckström, 1988b) can be seen as a first step in this direction. Axiom 2.6 expresses that no feature can be defined in both the prevail-condition and the pre-condition (and thus implicitly also in the post-condition) of an action. This conforms to the previous discussion of the different purposes of pre- and post-conditions and the prevail-condition respectively. Finally, axiom 2.7 assures that the pre-, post- and prevail-conditions are the only properties of an action type so that two distinct action types must differ in at least one of these conditions.

Example 2.2 Given the domains in example 2.1, we let $\mathcal{H} = \{h_1, h_2, h_3, h_4\}$ with b, e and f defined as in table 1.

We observe that h_1 violates axioms 2.3 and 2.4, and h_2 violates axioms 2.5 and 2.6. Neither h_3 nor h_4 contradicts any of the axioms, but, assuming that $h_3 \neq h_4$, the set $\mathcal{H}$ contradicts axiom 2.7 since h_3 and h_4 agree on all conditions. □

Since two actions of the same type are only different occurrences of the same action type, we only need some identification making these occurrences unique. Hence, an action consists of an action type and a unique *label*, the latter being the identification making this particular action unique. We also let an action 'inherit' the conditions from its associated action type.

Definition 2.8

1. $\mathcal{L}$ is an infinite set of *action labels*.
2. 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 first components (same labels). An element of a set of actions is referred to as an *action*.
3. If $\mathcal{A}$ is a set of actions we define two functions: $label : \mathcal{A} \to \mathcal{L}$ and $type : \mathcal{A} \to \mathcal{H}$ s.t. if $\langle l, h \rangle \in \mathcal{A}$ then $label(\langle l, h \rangle) = l$ and $type(\langle l, h \rangle) = h$.
4. The function *type* is generalized to sets of actions in the following way: $type(\mathcal{A}) = \{type(a) \mid a \in \mathcal{A}\}$
5. If $\mathcal{A}$ is a set of actions then we also extend the functions b, e and f s.t. $b(a) = b(type(a))$, $e(a) = e(type(a))$ and $f(a) = f(type(a))$ for all $a \in \mathcal{A}$.

□

Example 2.3 Let $\mathcal{L}$ be the natural numbers and $\mathcal{H} = \{h_1, h_2, h_3\}$, then $\{\langle 1, h_1\rangle, \langle 2, h_2\rangle, \langle 3, h_1\rangle\}$ is a set of actions, but $\{\langle 1, h_1\rangle, \langle 1, h_2\rangle, \langle 2, h_3\rangle\}$ is not a set of actions. □

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 *non-linear* if it is partially ordered. A non-linear plan is a *parallel plan* if its unordered actions can be executed in parallel without interfering with each other. That two actions are unordered in a non-parallel non-linear plan only means that they can be executed in either order but not in parallel, that is, such a plan must always be strengthened to a linear plan when executing it. Furthermore, a plan is minimal if there is no other plan solving the same problem using fewer actions.

The basic concept behind our formal definition of plans is the relation $\longmapsto$ which expresses how a sequence of actions can take us from one state to another. The basic concept behind parallel plans is the notion of independence; two actions are said to be *independent* iff they can be executed in parallel without interfering with each other. 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 thus avoided.

Definition 2.9
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 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 $1 \leq k \leq n$.

□

We will usually write $s \overset{a}{\longmapsto} s'$ as an abbreviation for $s \overset{\{a\},\varnothing}{\longmapsto} s'$. We will also frequently violate that $\sigma \subseteq \Psi^2$ and implicitly understand the restriction of σ to Ψ^2.

Definition 2.10 Assuming that $\Psi \subseteq \mathcal{L} \times \mathcal{H}$ is a set of actions, $\rho \subseteq \Psi^2$ and $s_o, s_\star \in \mathcal{S}$ we define:

1. $\langle \Psi, \rho \rangle$ is a *linear plan* from s_o to $s_\star$ iff ρ is a total order on Ψ and $s_o \overset{\Psi,\rho}{\longmapsto} s_\star$
2. $\langle \Psi, \rho \rangle$ is a *non-linear plan* from s_o to $s_\star$ iff ρ is a partial order on Ψ and $\langle \Psi, \sigma \rangle$ is a linear plan for any total order σ on Ψ s.t. $\rho \subseteq \sigma$.

□

Since the non-linear plans include the linear plans we will often write *plan* instead of non-linear plan.

Definition 2.11 A plan $\langle \Psi, \rho \rangle$ from s_o to $s_\star$ s.t. $type(\Psi) \subseteq \mathcal{H}$ is *minimal* w.r.t $\mathcal{H}$ iff there is no other plan $\langle \Phi, \sigma \rangle$ from s_o to $s_\star$ s.t. $type(\Phi) \subseteq \mathcal{H}$ and $|\Phi| < |\Psi|$. □

We will usually only say that a plan is minimal and understand the set $\mathcal{H}$ implicitly from the context.

Definition 2.12 Two actions a and a' are *independent* iff, for all $i \in \mathcal{M}$, all of the following conditions hold:

1. $b(a)[i] = u$ or $b(a')[i] = u$,
2. $b(a)[i] = u$ or $f(a')[i] = u$,
3. $b(a')[i] = u$ or $f(a)[i] = u$ and
4. $f(a)[i] \sqsubseteq f(a')[i]$ or $f(a')[i] \sqsubseteq f(a)[i]$

□

Definition 2.13 A non-linear plan $\langle \Phi, \rho \rangle$ from s_o to $s_\star$ is a *parallel* plan iff all pairs of actions $a, a' \in \Phi$ s.t. neither $a \rho a'$ nor $a' \rho a$ are independent. □

Definition 2.14 A *planning problem* is a tuple $\langle \mathcal{M}, \mathcal{S}_1, \ldots, \mathcal{S}_{|\mathcal{M}|}, \mathcal{H}, s_o, s_\star \rangle$ where $\mathcal{M}$ is a set of feature indices, $\mathcal{S}_1, \ldots, \mathcal{S}_{|\mathcal{M}|}$ are domains, $\mathcal{H}$ is a set of action types s_o is the initial state and $s_\star$ is the goal state. The planning problem is to find a set of actions Ψ and a partial order ρ on Ψ s.t. $type(\Psi) \in \mathcal{H}$ and $\langle \Psi, \rho \rangle$ is a plan from s_o to $s_\star$. □

The set $\mathcal{H}$ and the states s_o and $s_\star$ must of course be compatible with the choice of $\mathcal{M}$ and $\mathcal{S}_1 \ldots \mathcal{S}_{|\mathcal{M}|}$.

3 Classes of Planning Problems

The class of planning problems definable in our ontology so far is referred to as the *SAS* (Simplified Action Structures) class.

Definition 3.1 The class of planning problems with $\mathcal{M}, S_1, \ldots, S_{|\mathcal{M}|}$, and $\mathcal{H}$ as defined in section 2 and with no further restrictions than those mentioned in that section is referred to as the *SAS class.* □

We 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 and a planning problem is binary if all its domains are binary. A set of action types is *unary* if all its action types change exactly one feature. A set of action types is *post-unique* if it does not have two distinct action types changing the same feature to the same value. A set of action types is *single-valued* if whenever two different of its action types define the same feature in their prevail-conditions, they also define the same value for this feature. A set of action types is *prevail minimal* if it does not have two different action types that differ only in their prevail-conditions and the prevail-condition of one of these is subsumed by the prevail-condition of the other.

The remainder of this article studies the subclass of the SAS class that exhibits all of these restrictions. This subclass is called the *SAS-PUBS* class where PUBS is an acronym for Post-unique, Unary, Binary and Single-valued. Prevail-minimality is implied by these four restrictions. The practical implications of the restrictions are discussed in section 6. It could, however, be pointed out already that single-valuedness is probably the most serious restriction and it is crucial for the results in this article.

Definition 3.2 The domain $\mathcal{S}_i$, where $i \in \mathcal{M}$, is *binary* iff $|\mathcal{S}_i| = 2$. The state space $\mathcal{S}$ is binary iff $\mathcal{S}_i$ is binary for all $i \in \mathcal{M}$. □

Definition 3.3 An action type $h \in \mathcal{H}$ is *unary* iff $dim(b(h))$ is a singleton. A set of action types $\mathcal{H}$ is unary if all actions in $\mathcal{H}$ are unary. □

Definition 3.4 A set of action types $\mathcal{H}$ is *post-unique* iff

$$\forall h, h' \in \mathcal{H}(\exists i \in \mathcal{M}(e(h)[i] = e(h')[i] \neq u) \rightarrow h = h')$$

□

Definition 3.5 A set $\mathcal{H}$ of action types is *single-valued* iff

$$\exists c \in \mathcal{S} \forall h \in \mathcal{H}(f(h) \sqsubseteq c)$$

□

Definition 3.6 A set $\mathcal{H}$ of action types is *prevail minimal* iff

$$\forall h, h' \in \mathcal{H}(b(h) = b(h') \wedge e(h) = e(h') \wedge f(h) \sqsubseteq f(h') \rightarrow h = h')$$

□

Theorem 3.7 If $\mathcal{H}$ is unary and post-unique, then $\mathcal{H}$ is prevail minimal. □

Proof: Suppose $\mathcal{H}$ is unary and post-unique. Further suppose there are $h, h' \in \mathcal{H}$ s.t. $b(h) = b(h')$, $e(h) = e(h')$ and $f(h) \sqsubseteq f(h')$. By definition 3.3, there is some $i \in \mathcal{M}$ s.t. $e(h)[i] \neq u$, so $e(h)[i] = e(h')[i] \neq u$. Definition 3.4 give that $h = h'$, so we have $b(h) = b(h') \wedge e(h) = e(h') \wedge f(h) \sqsubseteq f(h') \rightarrow h = h'$, which is the definition of prevail minimality. □

Definition 3.8 A planning problem $\langle \mathcal{M}, \mathcal{S}_1, \ldots, \mathcal{S}_{|\mathcal{M}|}, \mathcal{H}, s_o, s_\star \rangle$ is in the SAS-PUBS class iff it is SAS, $\mathcal{S}_i$ is binary for all $i \in \mathcal{M}$ and $\mathcal{H}$ is unary, post-unique and single-valued[2]. □

4 Planning for SAS-PUBS Problems

This section presents some results about plans for SAS-PUBS problems, how to find such plans andthe complexity of finding such plans. The first subsection presents some auxiliary definitions needed. The second subsection presents a criterion for the existence of minimal parallel plans for the SAS-PUBS class and a proof that this criterion is correct. The third subsection presents an algorithm for finding such plans and a correctness proof for the algorithm. The section concludes with a complexity analysis of the algorithm.

The main definitions are 4.5 and 4.6 stating the existence criterion for parallel minimal SAS-PUBS plans and algorithm 4.1 presenting an algorithm for finding such plans. The main theorems are 4.25, proving the correctness of the existence criterion, 4.36 proving that the algorithm finds a parallel minimal plan iff there is a plan at all and 4.38, 4.41 and 4.42 stating some complexity results for the algorithm.

The reader is advised to first take a look at algorithm 4.1 and then go through the example in section 5 before reading this section more carefully. The more practically oriented reader could skip the whole section and read only the definitions and theorems mentioned above.

[2]Note that the SAS-PUBS class is implicitly prevail-minimal.

4.1 Auxiliary Definitions

We start by defining a few useful concepts. We say that an action *affects* those features that it changes. For binary domains, we introduce the concept of *inverse* such that the inverse of a defined value is the other defined value of the domain while the undefined and contradictory values are not affected by inversion. The inverse of a state is a state where all features are the inverses of the corresponding features in the first state. We also talk about the inverse of an action meaning an action with pre- and post-conditions being the inverses of those of the first action. Given a planning problem we also define set and reset actions. A *set action* for a certain feature is an action that changes that feature from its value in the initial state to the inverse value and a *reset action* for the same feature is an action that changes the value back to its value in the initial state. Finally, a *set/reset pair* for a certain feature is pair consisting of a set action and a reset action for that feature.

Definition 4.1

1. An action type $h \in \mathcal{H}$ *affects* the feature $i \in \mathcal{M}$ iff $i \in dim(b(h))$.
2. An action a affects the feature $i \in \mathcal{M}$ iff $type(a)$ affects i.
3. If Ψ is a set of actions and $i \in \mathcal{M}$, then the set $\Psi[i]$ denotes the set of all $a \in \Psi$ s.t. a affects i.

$\square$

Definition 4.2
If $\mathcal{S}_i$ is binary and $x \in \mathcal{S}_i$ then $\overline{x}$, the *inverse* of x, is defined s.t. $\overline{x} \in \mathcal{S}_i$ and $\overline{x} \neq x$. The inverse is extended to $\mathcal{S}_i^+$ s.t. $\overline{u} = u$, $\overline{k} = k$ and $\overline{x}$ is defined as above for $x \in \mathcal{S}_i$. The inverse is further extended to states s.t., for $s \in \mathcal{S}_i^+$, $\overline{s}[i] = \overline{s[i]}$ for all $i \in \mathcal{M}$. $\square$

Definition 4.3
Assuming $\mathcal{S}_i$ is binary for all $i \in \mathcal{M}$ and given a set $\mathcal{A}$ of actions, if for any action $a \in \mathcal{A}$ there is a unique action $a' \in \mathcal{A}$ s.t. $b(a') = \overline{b(a)}$, and thus implicitly $e(a') = \overline{e(a)}$, then a' is is called the *inverse* of a and is denoted $\overline{a}$. $\square$

Definition 4.4
Given a planning problem $\langle \mathcal{M}, \mathcal{S}_1, \ldots, S_{|\mathcal{M}|}, \mathcal{H}, s_o, s_\star \rangle$ where S_i is binary for all

$i \in \mathcal{M}$ we say that an action a s.t. *type*$(a) \in \mathcal{H}$ and $b(a)[i] = s_o[i]$ is a *set action* for feature i. An action a s.t. *type*$(a) \in \mathcal{H}$ and $e(a)[i] = s_o[i]$ is called a *reset action* for feature i, and a pair of actions a, a' s.t. a is a set action for i and a' is a reset action for i is called a *set/reset pair* for i. □

4.2 Existence of SAS-PUBS Plans

We first define the set $\Delta(s_o, s_\star)$ of necessary and sufficient actions for a minimal plan solving a SAS-PUBS problem, and we then define the execution order δ_Δ on this set. These two definitions together form the existence criterion for parallel minimal plans mentioned in the introduction to this section. The rest of the subsection is devoted to proving that the tuple $\langle \Delta(s_o, s_\star), \delta_\Delta \rangle$ indeed is a parallel minimal plan from s_o to $s_\star$ and that this tuple exists iff there is a plan at all from s_o to $s_\star$.

Definition 4.5 Given a SAS-PUBS problem, the set $\Delta(s_o, s_\star)$ of necessary and sufficient actions for a plan from s_o to $s_\star$ is recursively defined as follows:

1. $\mathcal{A} = \{\langle g(h), h \rangle \mid h \in \mathcal{H}\}$ where $g : \mathcal{H} \to \mathcal{L}$ is an arbitrary injection.
2. (a) For each $i \in \mathcal{M}$ s.t. $s_o[i] \neq s_\star[i]$ there is exactly one action $a \in \mathcal{A}$ s.t. $b(a)[i] = s_o[i]$, $e(a)[i] = s_\star[i]$ and $a \in P_0$.
   No other actions belong to P_0.
   (b) $T_0 = P_0$
   (c) $A_0 = \mathcal{A} - P_0$
3. For $k \geq 0$:
   (a) For each $a \in P_k$ and for each $i \in \mathcal{M}$ if $f(a)[i] \not\sqsubseteq s_o[i]$ and there is no $a' \in T_k$ s.t. $e(a')[i] = f(a)[i]$ then there are exactly two actions $a_1, a_2 \in A_k$ s.t. $b(a_1)[i] = s_o[i]$, $e(a_1)[i] = f(a)[i] = b(a_2)[i]$, $e(a_2)[i] = s_\star[i]$ and $a_1, a_2 \in P_{k+1}$.
   No other actions belong to P_{k+1}.
   (b) $T_{k+1} = T_k \cup P_{k+1}$
   (c) $A_{k+1} = A_k - P_{k+1}$
4. $\Delta(s_o, s_\star) = \cup_{k=0}^{\infty} P_k$

□

The first part of definition 4.5 says that A is a set containing exactly one action of each type in $\mathcal{H}$, it will turn out later that a minimal SAS-PUBS plan contains at most one action of every type. The set $\Delta(s_o, s_\star)$ is then recursively defined as the union of an infinite sequence $P_0, P_1, \ldots$ of sets of actions. The set P_0 contains exactly those actions that are required in order to change those features that differ between s_o and $s_\star$. The sets $P_1, P_2, \ldots$ contain set/reset pairs for those features that are not to be changed permanently but have to be changed temporarily to fulfil the prevail-conditions of other actions in the plan. In other words, if P_k contains an action a whose prevail-condition for some feature i is not fulfilled by any action in $P_0 \cup \ldots \cup P_k$ then P_{k+1} contains a set/reset pair for the i:th feature. This accomplishes that the i:th feature is temporarily set to the value required by the prevail-condition of a and then changed back to its original value again, which must also be the value of feature i in $s_\star$ since there was no action in P_0 affecting this feature. The set $\Delta(s_o, s_\star)$ is defined as the union of all P_k.

We define the execution order δ_Δ for the actions in $\Delta(s_o, s_\star)$ using the orders γ and η. These are defined s.t. $a\gamma a'$ if a sets some feature to the value required by the prevail-condition of a' and we say that a 'enables' a'. We also define that $a'\eta a$ if a changes some feature from the value required by the prevail-condition of a' to some other value and we say that a 'disables' a'. The order δ is defined as the transitive closure of the union of the orders γ and η and if $a\delta a'$ we say that a 'precedes' a'.

Definition 4.6 Suppose Φ is a set of actions or action types, then the relation δ_Φ on Φ is defined as:

1. $\forall a, a' \in \Phi(a\gamma_\Phi a' \leftrightarrow \exists i \in \mathcal{M}(e(a)[i] = f(a')[i] \neq u))$
2. $\forall a, a' \in \Phi(a\eta_\Phi a' \leftrightarrow \exists i \in \mathcal{M}(f(a)[i] = b(a')[i] \neq u))$
3. $\theta_\Phi = \gamma_\Phi \cup \eta_\Phi$
4. $\delta_\Phi = \theta_\Phi^+$

□

Below follows the proof that the the above definitions characterize exactly the parallel minimal plans, which is stated in theorem 4.25. To ease the burdens of notation somewhat we will usually write Δ and implicitly understand this as $\Delta(s_o, s_\star)$ and we will also omit the subscripts to the relations γ, η, θ and δ if it is clear from context which set they refer to. We will furthermore also implicitly understand that all plans are plans from s_o to $s_\star$ unless otherwise stated.

Definition 4.7 A function $r : \mathcal{L} \to \mathcal{L}$ is called a *relabelling* iff it is a permutation on $\mathcal{L}$. If r is a relabelling it is extended to also be a function $r : \mathcal{L} \times \mathcal{H} \to \mathcal{L} \times \mathcal{H}$ defined as $r(\langle l, h\rangle) = \langle r(l), h\rangle$, and it is further extended to be a function $r : 2^{\mathcal{L}\times\mathcal{H}} \to 2^{\mathcal{L}\times\mathcal{H}}$ defined for sets of actions as $r(A) = \{r(a) \mid a \in A\}$. □

Definition 4.8 Two sets of actions A and A' are *isomorphic* iff there exists a bijection $g : A \to A'$ s.t. $type(a) = type(g(a))$ for $a \in A$. □

Theorem 4.9 If A is a set of actions and r is a relabelling then $r(A)$ is a set of actions isomorphic to A. □

Proof: Suppose A is a set of actions, then $A \subseteq \mathcal{L} \times \mathcal{H}$ and by definition also $r(A) \subseteq \mathcal{L} \times \mathcal{H}$. Since A is a set of actions, all $a \in A$ have unique labels, so, since r is a permutation on $\mathcal{L}$, all $a \in r(A)$ also have unique labels. It follows, by definition 2.8, that $r(A)$ is a set of actions.
To prove that $r(A)$ is isomorphic to A we first define an inverse r^{-1} to r as follows: $r^{-1}(r(l)) = l$, $r^{-1}(\langle l, h\rangle) = \langle r^{-1}(l), h\rangle$ and $r^{-1}(A) = \{r^{-1}(a) \mid a \in A\}$. It is easily verified that r^{-1} exists if r exists, so r is a bijection and it follows that $r(A)$ is isomorphic to A. □

Lemma 4.10 If $\langle \Psi, \rho\rangle$ is a plan and Δ exists then there is a relabelling r s.t. $r(\Delta) \subseteq \Psi$. □

Proof: We prove that for all actions $a \in \Delta$ we can choose a unique action $a' \in \Psi$ s.t. $type(a) = type(a')$. Using the fact that $\Delta = \cup_{k=0}^{\infty} P_k$ we make a proof by induction on k.

Basis: Let $D = \{i \in \mathcal{M} \mid s_o[i] \neq s_\star[i]\}$. By definition 4.5 P_0 contains exactly one action for each $i \in D$ and no other actions, so $|P_0| = |D|$. Let $a_1, \ldots, a_{|D|}$ be an enumeration of P_0. Since $\mathcal{H}$ is unary, Ψ must contain at least one action for each $i \in D$ in order to change s_o to $s_\star$. Select one such action from Ψ for each $i \in D$ and let $a'_1, \ldots, a'_{|D|}$ be an enumeration of these actions s.t., for $1 \leq j \leq |D|$, a'_j and a_j affect the same feature. Since $\mathcal{H}$ is post-unique there are no alternative ways to change $s_o[i]$ to $s_\star[i]$ for any $i \in D$, so $type(a_j) = type(a'_j)$ for $1 \leq j \leq |D|$. We define a relabelling r_0 s.t. $r_0(label(a_j)) = label(a'_j)$ for $1 \leq j \leq |D|$ and $r_0(a)$ is undefined for $a \notin P_0$. It follows that $type(r_0(a)) = type(a)$ for all $a \in P_0$ and therefore also that $r_0(P_0) \subseteq \Psi$.

Induction: Suppose there is a relabelling r_j s.t. $r_j(P_j) \subseteq \Psi$ for $j > 1$. By definition 4.5, P_{j+1} is either empty or consists of set/reset pairs. The case where $P_{j+1} = \varnothing$ is trivial. For the other case, let $a_{11}, a_{12}, a_{21}, a_{22}, \ldots, a_{n1}, a_{n2}$ be an enumeration of P_{j+1} s.t., for $1 \leq m \leq n$, a_{m1}, a_{m2} is a set/reset pair for some unique feature $i \in \mathcal{M}$. For each m, a_{m1} is in P_{j+1} because of some action $a_m \in P_j$ s.t. $f(a_m)[i] \not\sqsubseteq s_o[i]$ and there is no action $a'_m \in T_j$ s.t. $e(a'_m)[i] = f(a_m)[i]$. The action a_{m2} is in P_{j+1} for the same reason, and with purpose of resetting feature i to assure $s_o[i] = s_\star[i]$. By the induction hypothesis, $r_j(P_j) \subseteq \Psi$ and thus also $r_j(a_m) \in \Psi$. Hence there must be two actions $a'_{m1}, a'_{m2} \in \Psi$ s.t. $e(a'_{m1})[i] = f(r_j(a_m))[i] = b(a'_{m2})[i]$ in order to fulfil the prevail-condition of $r_j(a_m)$ and to assure that $s_o[i] = s_\star[i]$. Since $\mathcal{S}$ is binary and $\mathcal{H}$ is post-unique, $type(a_{m1}) = type(a'_{m1})$ and $type(a_{m2}) = type(a'_{m2})$. It is thus possible to define a relabelling r_{j+1} s.t. $r_{j+1}(a_{m1}) = a'_{m1}$ and $r_{j+1}(a_{m2}) = a'_{m2}$ for $a_{m1}, a_{m2} \in P_{j+1}$, $r_{j+1}(a) = r_j(a)$ for $a \in T_j$ and r_{j+1} is otherwise undefined. Obviously, $type(r_{j+1}(a)) = type(a)$ for $a \in P_{j+1}$ and it follows that $r_{j+1}(P_{j+1}) \subseteq \Psi$ which proves the induction step.

Now, for $k \geq 0$, $r_k(P_k) \subseteq \Psi$ for P_k as defined in definition 4.5 and r_k as defined above. We define $r_\infty = \cup_{k=0}^{\infty} r_k$. Since, for $k > 0$, r_{k+1} always agree with r_k on arguments in T_k and r_k is always undefined for arguments not in T_{k+1}, it follows that r_∞ is a function. Furthermore, since all r_k are relabellings, r_∞ is also a relabelling. Consequently, $r_\infty(P_k) \subseteq \Psi$ for $k \geq 0$ and, since $\Delta = \cup_{k=0}^{\infty} P_k$, it follows that $r_\infty(\Delta) \subseteq \Psi$, which proves the lemma. □

Corollary 4.11 If $\langle \Psi, \rho \rangle$ is a plan and Δ exists then there is a relabelling r s.t. $\Delta \subseteq r(\Psi)$. □

Proof: We know from theorem 4.10 that there is a relabelling r' s.t. $r'(\Delta) \subseteq \Psi$. We know from the proof of lemma 4.9 that all relabellings have an inverse, so we let r'^{-1} be the inverse of r'. Obviously $\Delta \subseteq r'^{-1}(\Psi)$ □

Lemma 4.12 If there is a plan then Δ exists. □

Proof: Suppose that there is a plan $\langle \Psi, \rho \rangle$ and also suppose that there is no set Δ fulfilling definition 4.5. The non-existence of Δ can be for either of two reasons; either the set P_0 does not exist or there is a $k > 0$ s.t. P_k does not exist. Suppose that P_0 does not exist. The only possible reason for this is that there is an $i \in \mathcal{M}$ s.t. $s_o[i] \neq s_\star[i]$ but there is no $a \in \mathcal{A}$ s.t. $b(a)[i] = s_o[i]$ and $e(a)[i] = s_\star[i]$. It follows from the construction of $\mathcal{A}$ that for each $h' \in \mathcal{H}$ there is a unique $a' \in \mathcal{A}$ s.t. $type(a') = h'$. Consequently, there can be no $h \in \mathcal{H}$ s.t. $b(h)[i] = s_o[i]$ and $e(h)[i] = s_\star[i]$, and, since $\mathcal{H}$ is post-unique, there is no other $h'' \in \mathcal{H}$ s.t. $e(h'')[i] = s_\star[i]$. However, since $\langle \Psi, \rho \rangle$ is a plan from s_o to $s_\star$ and $s_o[i] \neq s_\star[i]$ there must be an action $a'' \in \Psi$ s.t. $e(a'')[i] = s_\star[i]$ and therefore also a $h''' \in \mathcal{H}$ s.t. $e(h''')[i] = s_\star[i]$. This is a contradiction, so P_0 must exist. The proof for the existence of P_k for $k > 0$ is analogous. This means that P_k exists for all $k \geq 0$ and, by definition 4.5, also Δ must exist. □

Lemma 4.13 If $\langle \Psi, \rho \rangle$ is a plan from s_o to $s_\star$ and there is a non-empty set $\Phi \subseteq \Psi$ s.t. $e(a) \sqsubseteq s_\star$ for all $a \in \Phi$, then there are two states $s, s' \in \mathcal{S}$ and an action $a' \in \Psi$ s.t. $s \overset{a'}{\longmapsto} s'$, $e(a) \sqsubseteq s'$ for all $a \in \Phi$ and $type(a') = type(a)$ for some $a \in \Phi$. □

Proof: Let $I_\Phi = \{i \in \mathcal{M} \mid a \in \Phi \wedge e(a)[i] \neq u\}$, and let $\mathcal{S}'$ be the set of all states $s \in \mathcal{S}$ s.t. $s[i] = s_\star[i]$ for all $i \in I_\Phi$. Since $\Phi \subseteq \Psi$ is non-empty, there must be some action $a \in \Psi$ and two states $s, s' \in \mathcal{S}$ s.t. $s \notin \mathcal{S}'$, $s' \in \mathcal{S}'$ and $s \overset{a}{\longmapsto} s'$. That $e(a') \sqsubseteq s'$ for all $a' \in \Phi$ is immediate, and, since a obviously affects some $j \in I_\Phi$ and $\mathcal{H}$ is post-unique, it also follows that $type(a) = type(a')$ for some $a' \in \Phi$. □

Lemma 4.14 If there is a plan $\langle \Psi, \rho \rangle$ from s_o to $s_\star$ and Δ exists then δ is a partial order. □

Proof: Suppose that δ is not partially ordered. Then δ is either not antireflexive or not antisymmetric since it is transitive by definition. However, non-antisymmetry implies non-irreflexivity so δ is either not antireflexive but antisymmetric, or not antisymmetric.

1. Suppose that δ is not antireflexive but antisymmetric. Then there must be an action $a \in \Delta$ s.t. $a \delta a$, and, by antisymmetry and transitivity, we get $a \theta a$. This means that either $a \gamma a$ or $a \eta a$, that is, either $e(a)[i] = f(a)[i] \neq u$ or $f(a)[i] = b(a)[i] \neq u$ for some $i \in \mathcal{M}$ both of which are contradicted by axioms 2.4 and 2.6.

2. Suppose δ is not antisymmetric. Then there are two different actions $a', a'' \in \Delta$ s.t. $a' \delta a''$ and $a'' \delta a'$. It follows from definition 4.6 that there is a sequence

$a_1, \ldots, a_n \in \Delta \subseteq r(\Psi)$ of actions s.t. $a_k \sigma a_{k+1}$ for $1 \leq k \leq n$ and $a_n \sigma a_1$ where $\sigma \subseteq \theta$ and r is a relabelling s.t. $\Delta \subseteq r(\Psi)$ (exists by corollary 4.11). There are now three cases: $\sigma \subseteq \eta$, $\sigma \subseteq \gamma$ or neither of these two.

(a) Suppose $\sigma \subseteq \eta$ so that $a_k \eta a_{k+1}$ for $1 \leq k \leq n$ and $a_n \eta a_1$. Once again, there are two cases; either $e(a_k) \sqsubseteq s_\star$ for $1 \leq k \leq n$ or not.

   i. Suppose $e(a_k) \sqsubseteq s_\star$ for $1 \leq k \leq n$. We know, by lemma 4.13, that there are two states $s, s' \in \mathcal{S}$ s.t. for some action $a \in \Psi$, $s' \overset{a}{\longmapsto} s$, $e(a_k) \sqsubseteq s$ for $1 \leq k \leq n$ and $type(a) = type(a_l)$ for some l s.t. $1 \leq l \leq n$. By assumption there is an m s.t. $a_l \eta a_m$, i.e. $f(a_l)[i] = b(a_m)[i] \neq u$ for some $i \in \mathcal{M}$. Furthermore, $f(a_l)[i] = f(a)[i] \sqsubseteq s[i]$, so $b(a_m)[i] = s[i]$. By hypothesis and axiom 2.4 we have $e(a_m)[i] = s[i]$, so $b(a_m)[i] = e(a_m)[i]$ which contradicts axiom 2.5.

   ii. Suppose that $e(a_l) \not\sqsubseteq s_\star$ for some l s.t. $1 \leq l \leq n$. It is obvious from definition 4.5 that $a_l \notin P_0$ and that a_l is a set action for some feature $i \in \mathcal{M}$. From the same definition it also follows that there is an action $a \in \Delta$ s.t. $e(a_l)[i] = f(a)[i] \neq u$. From the hypothesis we know that for some m s.t. $1 \leq m \leq n$ we have $a_m \eta a_l$ and thus also $f(a_m)[j] = b(a_l)[j] \neq u$ for some $j \in \mathcal{M}$. Unariness gives $i = j$, and by axiom 2.5 we get $b(a_l)[i] \neq e(a_l)[i]$, so $u \neq f(a)[i] \neq f(a_m)[i] \neq u$ which contradicts the single-valuedness of $\mathcal{H}$.

(b) The case where $\sigma \subseteq \gamma$ is analogous to the previous case.

(c) Suppose that neither $\sigma \subseteq \eta$ nor $\sigma \subseteq \gamma$, then there are k, l and m s.t. $1 \leq k, l, m \leq n$, $a_k \eta a_l$ and $a_l \gamma a_m$. Hence $f(a_k)[i] = b(a_l)[i] \neq u$ and $e(a_l)[j] = f(a_m)[j] \neq u$ for some $i, j \in \mathcal{M}$, but $\mathcal{H}$ is unary, so $i = j$. Now, axiom 2.5 gives $b(a_l)[i] \neq e(a_l)[i]$ so $f(a_k)[i] \neq f(a_m)[i]$ which contradicts the single-valuedness of $\mathcal{H}$.

□

Definition 4.15 In order to increase readability of the following proofs we define $\tilde{P} = \cup_{k=1}^{\infty} P_k$. □

Lemma 4.16 Given a feature $i \in \mathcal{M}$, a set Ψ of actions s.t. none of its actions affects i, and a total order σ on Ψ; if $s \overset{\Psi,\sigma}{\longmapsto} s'$ for some $s, s' \in \mathcal{S}$ then $s[i] = s'[i]$. □

Proof: Proof by induction over the size of Ψ.

Basis: Suppose $\Psi = \varnothing$, then, by def 2.9, $s \overset{\Psi,\sigma}{\longmapsto} s'$ iff $s = s'$, so $s[i] = s'[i]$.

Induction: Suppose the lemma holds for $|\Psi| \leq k$, and let Φ be any set of actions s.t. $|\Phi| = k + 1$ and Φ contains no actions affecting i. Now, let a be the last action in Φ, according to the order σ, and let $\Phi' = \Phi - \{a\}$. If $s \overset{\Phi,\sigma}{\longmapsto} s'$, then there must also be a state $s'' \in \mathcal{S}$ s.t. $s \overset{\Phi',\sigma}{\longmapsto} s''$ and $s'' \overset{a}{\longmapsto} s'$. It follows from the induction hypothesis that $s[i] = s''[i]$ and from definition 2.9 that $s''[i] = s'[i]$, so $s[i] = s'[i]$. Consequently, the lemma holds for all totally ordered sets of actions not affecting i. $\Box$

Lemma 4.17 For each $i \in \mathcal{M}$, one of three cases occur: $\Delta[i] = \varnothing$, $\Delta[i] = P_0[i] = \{a\}$ or $\Delta[i] = \tilde{P}[i] = \{a, \overline{a}\}$, where, in the two latter cases, a is set action for i. $\Box$

Proof: The case $\Delta[i] = \varnothing$ occurs when $s_o[i] = s_\star[i]$ and for all actions $a \in \Delta$, $f(a)[i] \sqsubseteq s_o[i]$. For the other cases, suppose that $\Delta[i] \neq \varnothing$, and let m be the minimal $k \geq 0$ s.t. $P_k[i] \neq \varnothing$. Suppose $m = 0$, then $P_0[i] \neq \varnothing$ so $s_o[i] \neq s_\star[i]$ and there is a set action a for i in P_0, and, by definition 4.5, $P_0[i] = \{a\}$. Now suppose that $m > 0$, then, by definition 4.5, $P_m[i] = \{a, \overline{a}\}$ where a is a set action for i. Furthermore, for all $k > m \geq 0$, $P_m \subseteq T_k$, so there is a set action a for i in T_k, and, by definition 4.5, $P_k[i] = \varnothing$. Hence, $\Delta[i] = P_m[i]$, so, when $\Delta[i] \neq \varnothing$, either $\Delta[i] = P_0[i] = \{a\}$ or $\Delta[i] = \tilde{P}[i] = \{a, \overline{a}\}$ where a is a set action for i. $\Box$

Lemma 4.18 If Δ exists and δ is a partial order on Δ, then there is a state $s \in \mathcal{S}$ s.t. $\langle \Delta, \delta \rangle$ is a plan from s_o to s. $\Box$

Proof: We prove that, for any total order σ s.t. $\delta \subseteq \sigma$, $\langle \Delta, \sigma \rangle$ is a linear plan, which amounts to proving $s_o \overset{\Delta,\sigma}{\longmapsto} s$.

Let $n = |\Delta|$ and let $a_1, \ldots, a_n$ be the actions in Δ as ordered under σ. We prove by induction on k that, for $1 \leq k \leq n$, there are $s_{k-1}, s_k \in \mathcal{S}$ s.t. $s_{k-1} \overset{a_k}{\longmapsto} s_k$.

Basis: Suppose $b(a_1) \sqcup f(a_1) \not\sqsubseteq s_o$, then either $b(a_1) \not\sqsubseteq s_o$ or $f(a_1) \not\sqsubseteq s_o$.

1. Suppose $b(a_1) \not\sqsubseteq s_o$, then there is an $i \in \mathcal{M}$ s.t. $u \neq b(a_1)[i] \neq s_o[i]$. $\mathcal{S}_i$ is binary, so, by axiom 2.5, we get $e(a_1)[i] = s_o[i]$. Hence, $a_1 \notin P_0$, which

means that $a_1 \in \tilde{P}$ and, furthermore, a_1 must be a reset action for i and, by definition 4.5, there is some action $a \in \Delta$ s.t. $f(a)[i] = b(a_1)[i]$. This gives $a\eta a_1$ which implies $a\delta a_1$ and also $a\sigma a_1$, which contradicts that a_1 is the first action in Δ under the order σ. Consequently, $b(a_1) \sqsubseteq s_o$.

2. Suppose $f(a_1) \not\sqsubseteq s_o$, then there is an $i \in \mathcal{M}$ s.t. $u \neq f(a_1)[i] \neq s_o[i]$. By definition 4.5 there must be an action $a \in \Delta$ s.t. $e(a)[i] = f(a_1)[i]$, which implies $a\gamma a_1$ and thus also $a\sigma a_1$. This contradicts that a_1 is the first action in Δ under σ, so $f(a_1) \sqsubseteq s_o$.

Since both $b(a_1)[i] \sqsubseteq s_o$ and $f(a_1)[i] \sqsubseteq s_o$ there must be some state $s_1 \in \mathcal{S}$ s.t. $s_o \stackrel{a_1}{\longmapsto} s_1$.

Induction: For $1 \leq k < n$, suppose that there are states $s_{k-1}, s_k \in \mathcal{S}$ s.t. $s_{k-1} \stackrel{a_k}{\longmapsto} s_k$, and also suppose that $b(a_{k+1}) \sqcup f(a_{k+1}) \not\sqsubseteq s_k$.

1. Suppose that $b(a_{k+1}) \not\sqsubseteq s_k$, then there is some $i \in \mathcal{M}$ s.t. $u \neq b(a_{k+1})[i] \neq s_k[i]$. There are now two cases:

   (a) Suppose $s_k[i] = s_o[i]$, then, by axiom 2.5 and binariness of $\mathcal{S}_i$, $e(a_{k+1})[i] = s_o[i]$, so a_{k+1} is a reset action for i and, since P_0 contains only set actions, $a_{k+1} \in \tilde{P}$. Hence, there must be an action $a \in \Delta$ s.t. $f(a)[i] = b(a_{k+1})[i]$, so $a\eta a_{k+1}$ and also $a\sigma a_{k+1}$. We further know, by lemma 4.17, that $\Delta[i] = \{\overline{a_{k+1}}, a_{k+1}\}$. Now, $e(\overline{a_{k+1}})[i] = f(a)[i]$, so $\overline{a_{k+1}}\sigma a_{k+1}$. It follows from the induction hypothesis and lemma 4.16 that $s_k[i] = e(\overline{a_{k+1}})[i]$, but $e(\overline{a_{k+1}})[i] = b(a_{k+1})[i]$, which contradicts the assumption.

   (b) Suppose $s_k[i] \neq s_o[i]$, then $b(a_{k+1})[i] = s_o[i]$ and a_{k+1} must be a set action for i. Definition 4.6 and lemma 4.17 give that there is no action $a \in \Delta$ s.t. $a\sigma a_{k+1}$ and a affects i, so lemma 4.16 give that $s_k[i] = s_o[i]$, which contradicts the assumption.

   Consequently, $b(a_{k+1}) \sqsubseteq s_k$.

2. Now suppose that $f(a_{k+1}) \not\sqsubseteq s_k$, which means that $u \neq f(a_{k+1})[i] \neq s_k[i]$ for some $i \in \mathcal{M}$. There are two cases:

   (a) Suppose $f(a_{k+1})[i] = s_o[i]$. Now, if there is some action $a \in \Delta$ s.t. $b(a)[i] = s_o[i] = f(a_{k+1})[i]$ then $a_{k+1}\eta a$ and thus also $a_{k+1}\sigma a$. Lemma 4.16 give $s_k[i] = s_o[i] = f(a_{k+1})[i]$, contradicting the assumption.

   (b) Suppose $f(a_{k+1})[i] \neq s_o[i]$, then, by definition 4.5, there is a set action $a \in \Delta$ for i s.t. $e(a)[i] = f(a_{k+1})[i]$, and thus also $a\gamma a_{k+1}$ and $a\sigma a_{k+1}$.

Either $a \in P_0$ and then $\Delta[i] = \{a\}$, or $a \in \tilde{P}$ and then $\Delta[i] = \{a, \overline{a}\}$. In the latter case, $b(\overline{a})[i] = f(a_{k+1})[i]$, so $a\eta a_{k+1}$ and thus also $a\sigma a_{k+1}$. In either case is a the only action that affects i and is ordered before a_{k+1}. By lemma 4.16 and induction hypothesis, $s_k[i] = e(a)[i] \neq s_o[i]$, so $f(a_{k+1})[i] = s_k[i]$, which contradicts the assumption.

Consequently, $f(a_{k+1}) \sqsubseteq s_k$.

Since both $b(a_{k+1}) \sqsubseteq s_k$ and $f(a_{k+1}) \sqsubseteq s_k$, there is a state s_{k+1} s.t. $s_k \overset{a_{k+1}}{\longmapsto} s_{k+1}$, which ends the induction step.

Putting $s = s_n$ concludes the proof. □

Lemma 4.19 If $\langle \Delta, \delta \rangle$ is a plan from s_o to s for some state $s \in \mathcal{S}$ then $s = s_\star$. □

Proof: We prove that if $\langle \Delta, \sigma \rangle$ is a plan from s_o to s for an arbitrary total order σ s.t. $\delta \subseteq \sigma$, then $s = s_\star$. This amounts to proving that if $s_o \overset{\Delta,\sigma}{\longmapsto} s$ then $s = s_\star$. We first define $D = \{i \in \mathcal{M} \mid s_o[i] \neq s_\star[i]\}$ and divide the proof into two parts, the first for features in D and the second for features not in D.

For the first part, we observe from definition 4.5 that for each $i \in D$ there is exactly one action in P_0 affecting i. For $i \in D$, lemma 4.17 give $\Delta[i] = P_0[i] = \{a\}$ where a is a set action for i. Let $\Phi^- = \{a' \in \Delta \mid a'\sigma a\}$ and $\Phi^+ = \{a' \in \Delta \mid a\sigma a'\}$. There must, by definition 2.9 be states $s_1, s_2 \in \mathcal{S}$ s.t. $s_o \overset{\Phi^-,\sigma}{\longmapsto} s_1$, $s_1 \overset{a}{\longmapsto} s_2$ and $s_2 \overset{\Phi^+,\sigma}{\longmapsto} s$, and, by lemma 4.16, $s[i] = s_2[i] = e(a)[i] = s_\star[i]$.

For the second part, we first observe that $\tilde{P} = \Delta - \cup_{i \in D}\Delta[i] = \cup_{i \notin D}\Delta[i]$, so $\Delta[i] \subseteq \tilde{P}$ for all $i \notin D$. It follows from lemma 4.17 that, for $i \notin D$, either $\Delta[i] = \varnothing$ or $\Delta[i] = \{a, \overline{a}\}$. Suppose $\Delta[i] = \varnothing$, then it is immediate from 4.5 that $s_\star[i] = s_o[i]$ and from lemma 4.16 that $s[i] = s_o[i]$, so $s[i] = s_\star[i]$. Now suppose that $\Delta[i] = \{a, \overline{a}\}$ where a is a set action for i. According to definition 4.5 there must be some action $a' \in \Delta$ s.t. $e(a)[i] = f(a')[i] = b(\overline{a})[i]$. Hence $a\gamma a'$ and $a'\eta\overline{a}$ from which follows that $a\delta\overline{a}$ and also $a\sigma\overline{a}$. We define $\Phi^- = \{a' \in \Delta \mid a'\delta a\}$, $\Phi = \{a' \in \Delta \mid a\sigma a' \wedge a'\sigma\overline{a}\}$ and $\Phi^+ = \{a' \in \Delta \mid \overline{a}\sigma a'\}$. Now, there must be states $s_1, s_2, s_3, s_4 \in \mathcal{S}$ s.t. $s_o \overset{\Phi^-,\sigma}{\longmapsto} s_1$, $s_1 \overset{a}{\longmapsto} s_2$, $s_2 \overset{\Phi,\sigma}{\longmapsto} s_3$, $s_3 \overset{\overline{a}}{\longmapsto} s_4$ and $s_4 \overset{\Phi^+,\sigma}{\longmapsto} s$. It follows from definition 2.9 and lemma 4.16 that $s[i] = s_o[i]$, and, by definition 4.5, also $s[i] = s_\star[i]$.

Consequently, $s[i] = s_\star[i]$ for all $i \in \mathcal{M}$, and thus also $s = s_\star$. □

Theorem 4.20 If Δ exists and δ is a partial order then $\langle \Delta, \delta \rangle$ is a plan. □

Proof: Immediate from lemmata 4.18 and 4.19. □

It is worth noticing that theorem 4.20 holds also if $\mathcal{H}$ is not single-valued.

Theorem 4.21 If there is a plan then Δ exists and $\langle \Delta, \delta \rangle$ is a plan. □

Proof: Immediate from lemmata 4.12, and 4.14 and from theorem 4.20. □

Theorem 4.22 If Δ exists and $\langle \Delta, \delta \rangle$ is a plan then $\langle \Delta, \delta \rangle$ is a minimal plan. □

Proof: Lemmata 4.10 and 4.12 gives that if $\langle \Psi, \rho \rangle$ is a plan then Δ exists and there is a relabelling function r s.t. $r(\Delta) \subseteq \Psi$. By theorem 4.9, Δ and $r(\Delta)$ are isomorphic so $|\Delta| = |r(\Delta)|$. It follows that $|\Delta| \leq |\Psi|$, so if Δ exists and there is a partial order σ on Δ s.t. $\langle \Delta, \sigma \rangle$ is a plan, then $\langle \Delta, \sigma \rangle$ is a minimal plan. □

Theorem 4.23 All minimal plans contains at most one action of each type in $\mathcal{H}$. □

Proof: It follows from lemma 4.10 and theorems 4.21 and 4.22 that all minimal plans contains the same number of each action type as Δ so the theorem follows from lemma 4.17. □

Theorem 4.24 If Δ exists then $\langle \Delta, \delta \rangle$ is a parallel plan. □

Proof: Suppose there is a pair of distinct actions $a, a' \in \Delta$ s.t. neither $a\delta a'$ nor $a'\delta a$ and a and a' are not independent. Definition 2.12 give that either of the following cases must apply.

1. Suppose $b(a)[i] \neq u$ and $b(a')[i] \neq u$, then lemma 4.17 give $a, a' \in \tilde{P}[i]$ and $a' = \overline{a}$. It follows from definition 4.5 that there is some $a'' \in \Delta$ s.t. $b(a)[i] = f(a'')[i] = e(a')[i]$ or $b(a')[i] = f(a'')[i] = e(a)[i]$. Suppose the first of these holds, then $a'\gamma a''$ and $a''\eta a$ so definition 4.6 give $a'\delta a$. The other case is symmetrical and result in $a\delta a'$ so the assumption is in violated in either case.
2. Suppose $b(a)[i] \neq u$ and $f(a')[i] \neq u$, then either $b(a)[i] = f(a')[i]$ or $e(a)[i] = f(a')[i]$ because of binariness, so either $a'\eta a$ or $a\gamma a'$. It follows by definition 4.6 that either $a'\delta a$ or $a\delta a'$ so the assumption is violated.
3. The case $b(a')[i] \neq u$ and $f(a)[i] \neq u$ is analogous to the previous case.
4. The case $f(a)[i] \not\sqsubseteq f(a')[i]$ and $f(a')[i] \not\sqsubseteq f(a)[i]$ is impossible because of single-valuedness.

Since neither case apply, there can be no such pair a, a' so definition 2.13 give that the lemma holds. □

Theorem 4.25 Δ exists and $\langle \Delta, \delta \rangle$ is a parallel minimal plan iff there is a plan at all. □

Proof: The if part follows from theorems 4.21, 4.22 and 4.24. The only-if part is immediate. □

4.3 Finding SAS-PUBS Plans

This section presents an algorithm that finds parallel minimal plans for SAS-PUBS problems according to the existence criterion stated in definitions 4.5 and 4.6. The presentation of the algorithm is followed by a correctness proof.

Definition 4.26 We assume that the following functions and procedures are available:

Insert(S,a) Inserts the action a into the set S.

Find(S,i,x) Searches the set S for an action a s.t. $b(a)[i] = x$. Returns a if found, otherwise returns **nil**.

Rfind(S,i,x) Like Find, but also removes a from S if it is found.

TransitiveClosure(R) Returns the transitive closure of the relation R.

□

Algorithm 4.1

Input: $\mathcal{M}$, a set of feature indices, A, a set containing exactly one action for each action 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(M :set of feature indices; A :set of actions; s_o, s_⋆ :state);
var
   i :feature index;
   a, a', a_1, a_2 :action;
   P, Q, D :set of actions;
   r :boolean matrix;

begin
   D := ∅;
   P := ∅;

   for i ∈ M do
      if s_o[i] ≠ s_⋆[i] then
         a :=Rfind(A, i, s_o[i]);
         if a ≠ nil then Insert(P, a);Insert(D, a)
         else fail
         end {if}
      end{if}
   end{for};

   while P ≠ ∅ do
      Q := ∅;
      for a ∈ P do
         for i ∈ M do
```

```
25              if f(a)[i] ⋢ s_o[i] then
26                a' := Find(D, i, s_o[i]);
27                if a' = nil then
28                  a_1 := Rfind(A, i, s_o[i]);
29                  a_2 := Rfind(A, i, f(a)[i]);
30                  if a_1 = nil or a_2 = nil then fail
31                  else Insert(Q, a_1);Insert(Q, a_2);
32                       Insert(D, a_1);Insert(D, a_2)
33                  end{if}
34                end{if}
35              end{if}
36            end{for}
37          end{for};
38          P := Q
39        end{while};
40
41        r := "|D| × |D| zero matrix";
42
43        for a ∈ D do
44          for a' ∈ D do
45            for i ∈ M do
46              if e(a)[i] = f(a')[i] ≠ u then r(a, a') := 1 end;
47              if b(a')[i] = f(a)[i] ≠ u then r(a, a') := 1 end
48            end{for}
49          end{for}
50        end{for};
51
52        r := TransitiveClosure(r)
53        return ⟨D, r⟩
54      end {Plan}
```

The first part of the algorithm, lines 12–19, compares the states s_o and $s_\star$ and for each feature that differs it searches A for an appropriate action to change this feature. If such an action is found it is removed from A and inserted into D and P, and otherwise the algorithm fails. Immediately after line 19, P corresponds to the set P_0 of definition 4.5. The next part, lines 21–39, finds the actions needed to satisfy the prevail-conditions of the actions in the plan. The variables are used so that the k:th time through the while loop $P = P_{k-1}$ and $Q = P_k$. The variable D is the union of all P_k:s so far. The while loop terminates as soon as $P = \varnothing$, that is $P_k = \varnothing$ after the k:th time through the loop. It is proven below that this is sufficient so no infinite chain of empty P_k:s need be constructed. $D = \Delta$ after the termination of the while loop. The for loops in lines 43–50 then goes through all pairs a, a' of actions in D and marks that ara' if $a\gamma a'$ or $a\eta a'$. Finally r is set to the transitive closure of itself, so r corresponds to the relation δ when the algorithm terminates.

The algorithm is not optimized since our goal has only been to prove tractability. However, it is obvious that the algorithm could be further optimized. For example, The relation r is probably better stored as an adjacency list. It is also worth noting that the post-conditions of the actions are never used in the algorithm. The rest of this subsection presents the correctness proof of the algorithm, which results in theorem 4.36.

Lemma 4.27 Throughout the execution of the algorithm, $A \subseteq \mathcal{A}$ and $D \subseteq \mathcal{A}$. □

Proof: Initially $A = \mathcal{A}$ and no actions are ever inserted into A, so clearly $A \subseteq \mathcal{A}$. Furthermore, all actions inserted into D are first found in A by the function *Rfind*, so $D \subseteq A$ and thus also $D \subseteq \mathcal{A}$. □

Lemma 4.28 If $P_n = \varnothing$ for some $n \geq 0$ then $\cup_{k=0}^{\infty} P_k = \cup_{k=0}^{n} P_k$. □

Proof: We prove by induction over k that $P_k = \varnothing$ for $k \geq n$. Basis: $P_n = \varnothing$ by assumption. Induction: If $P_k = \varnothing$ then $P_{k+1} = \varnothing$ by definition 4.5. Consequently, $P_k = \varnothing$ for $k \geq n$, so $\cup_{k=0}^{\infty} P_k = \cup_{k=0}^{n} P_k \cup \cup_{k=n+1}^{\infty} P_k = \cup_{k=0}^{n} P_k$. □

Lemma 4.29 If P_0 exists according to definition 4.5, then $P = P_0$, $D = T_0$ and $A = A_0$ at line 20 of the algorithm. □

Proof: This proof concerns the loop in line 12–19 of the algorithm. We first observe that *Rfind* is called at most once for each $i \in \mathcal{M}$, so, since $\mathcal{H}$ is unary, no action $a \in \mathcal{A}$ will be searched for in A more than once. Since actions can be deleted from A only by *Rfind*, no attempt will ever be made to delete an action already searched for in A. We will now prove that for each $a \in \mathcal{A}$ we have, at line 20, $a \in P$ iff $a \in P_0$. For the *if* case, suppose that $a \in P_0$. Hence, there must be an $i \in \mathcal{M}$ s.t. $s_o[i] \neq s_\star[i]$ and $b(a)[i] = s_o[i]$, so *Rfind* will be called to search for a in A. Since $a \in P_0 \subseteq \mathcal{A}$, initially $\mathcal{A} = A$, and, by the observation above, a has not been searched for earlier, we have $a \in A$. Consequently, a will be found and inserted into P. For the *only if* case, suppose that $a \notin P_0$, and $i \in \mathcal{M}$ is the feature affected by a. Now, since $a \notin P_0$, either $s_o[i] = s_\star[i]$ or $b(a)[i] \neq s_o[i]$, so either *Rfind* is never called to search for a, or one failed search for a is performed. In neither case is a inserted into P. Since P is initially empty and no actions are removed from P, it is obvious that $P = P_0$ in line 20. We furthermore observe that the actions inserted into D and deleted from A are exactly those actions inserted into P. Since, initially, $D = \varnothing$ and $A = \mathcal{A}$ and since nothing is inserted into A and nothing is deleted from D, we have $D = P = P_0 = T_0$ and $A = \mathcal{A} - P = \mathcal{A} - P_0 = A_0$ in line 20. □

Lemma 4.30 For $k \geq 0$, if P_{k+1} exists according to definition 4.5 and if $P = P_k$, $D = T_k$ and $A = A_k$ before the *k+1*:st iteration of the **while** loop in line 21–39 of the algorithm, then $P = P_{k+1}$, $D = T_{k+1}$ and $A = A_{k+1}$ after the *k+1*:st iteration of the loop. □

Proof: We first observe that the value of P is not changed until after the double **for** loop in line 23–37, so $P = P_k$ during this loop. Also $Q = \varnothing$ immediately before the double **for** loop. We further observe that no actions are deleted from Q or D and no actions are inserted into A, so $T_k \subseteq D$ and $A \subseteq A_k$ during the double **for** loop. We now prove that for all $a'' \in \mathcal{A}$, $a'' \in P_{k+1}$ iff $a'' \in Q$ immediately after the double **for** loop.

For the *if* case, suppose that $a'' \in Q$ at line 38, then a'' has been inserted into Q in some iteration of the double **for** loop. From the algorithm we get that either $b(a'')[i] = s_o[i]$ or $e(a'')[i] = s_o[i]$. The algorithm further gives that there is an action $a \in P$ s.t. $f(a)[i] \not\sqsubseteq s_o[i]$ and there is no action $a' \in D$ s.t. $e(a')[i] = f(a)[i]$. However, $P = P_k$ throughout the loop and $T_k \subseteq D$, so $a \in P_k$ and $a' \notin T_k$. Since P_{k+1} exists, there are, by definition 4.5, two actions $a_1, a_2 \in P_{k+1}$ s.t. $b(a_1)[i] = s_o[i]$ and $e(a_2)[i] = s_o[i]$. The set $\mathcal{A}$ contains at most one action of each type and $\mathcal{H}$ is post-unique, so, obviously, $a'' = a_1$ or $a'' = a_2$, and thus $a'' \in P_{k+1}$ in either case.

For the *only if* case, suppose that $a'' \in P_{k+1}$ and that $i \in \mathcal{M}$ is the feature affected by a''. Since $\mathcal{S}_i$ is binary, either $b(a'')[i] = s_o[i]$ or $e(a'')[i] = s_o[i]$. By definition 4.5, there is an action $a \in P_k$ s.t. $f(a)[i] \not\sqsubseteq s_o[i]$ and there is no action $a' \in T_k$ s.t. $e(a')[i] = f(a)[i]$. Now, let m be the number such that the value of the loop variables of the double **for** loop are a and i respectively during the m:th iteration of the double **for** loop. Such an m exists since $a \in P_k$ and $P = P_k$ during the loop. It follows that $f(a)[i] \not\sqsubseteq s_o[i]$ in the m:th iteration, so *Find* is called to search D for an action a' s.t. $b(a')[i] = s_o[i]$. This search either succeeds or fails.

Suppose it fails, then *Rfind* is called to search A for two actions a_1 and a_2 s.t. $b(a_1)[i] = s_o[i]$ and $b(a_2)[i] = f(a)[i]$. Since P_{k+1} exists, definition 4.5 give that there are two actions $a'_1, a'_2 \in A_k$ s.t $type(a_1) = type(a'_1)$ and $type(a_2) = type(a'_2)$. By construction, $\mathcal{A}$ contains at most one action of each type, so $a' = a_1 = a'_1$ and $a_2 = a'_2$, and, hence, $a', a_1, a_2 \in A_k$. Since the search for a' in D failed, $a' \in A_k$, $A = A_k$ immediately before the double **for** loop, and no actions are deleted from A without being inserted into D, $a' \in A$ immediately before the m:th iteration. Furthermore, either both a_1 and a_2 are deleted from A, or none of them is, so $a_2 \in A$ immediately before the m:th iteration. Consequently, the search for a_1 and a_2 in A succeeds, and these actions are thus also inserted into Q, so $a_1, a_2 \in Q$ in line 38. Now suppose that the search for a' in D succeeds. Since $a' \notin T_k$ and $T_k \subseteq D$, $a' = a_1$ must have been inserted into D in the l:th iteration of the double **for** loop for some l s.t. $1 \leq l < m$. Hence, also a_2 has been inserted into Q in the l:th iteration, so $a_1, a_2 \in Q$ in line 38. In either case we have $a_1, a_2 \in Q$ in line 38 and since $a'' = a_1$ or $a'' = a_2$ we also have $a'' \in Q$ in line 38.

Consequently, $a'' \in P_{k+1}$ iff $a'' \in Q$ at line 38, so $P = Q = P_{k+1}$ immediately after the $k{+}1$:st iteration of the **while** loop. Furthermore, the actions inserted into Q are exactly those actions inserted into D and deleted from A, so $D = T_k \cup Q = T_k \cup P_{k+1} = T_{k+1}$ and $A = A_k - Q = A_k - P_{k+1} = A_{k+1}$ after the $k{+}1$:st iteration of the **while** loop. □

Lemma 4.31 If Δ exists, then $D = \Delta$ after line 39 of the algorithm. □

Proof: If Δ exists, then P_k exists for all $k \geq 0$. Using lemma 4.29 as basis and lemma 4.30 as induction step it is easily proven by induction that $P = P_k$, $D = T_k$ and $A = A_k$ after the k:th iteration of the loop in line 21–39. Furthermore, the loop terminates as soon as $P = \varnothing$. Let m be the smallest k s.t. $P_k = \varnothing$, then the loop terminates after the m:th iteration, where we understand the case $m = 0$ as the case where the loop does not iterate at all. Lemma 4.28 and definition 4.5 give that $D = T_k = \cup_{k=0}^{m} P_k = \cup_{k=0}^{\infty} P_k = \Delta$. □

Lemma 4.32 If P_0 does not exist, then the algorithm fails before line 20. □

Proof: If P_0 does not exist, this must be because there is an $i \in \mathcal{M}$ s.t. $s_o[i] \neq s_\star[i]$, but there is no action $a \in \mathcal{A}$ s.t. $b(a)[i] = s_o[i]$ and $e(a)[i] = s_\star[i]$.

Since $s_o[i] \neq s_\star[i]$, the *Rfind* call in line 14 will search for an action $a' \in A$ s.t. $b(a')[i] = s_o[i]$, but, since $A \subseteq \mathcal{A}$, $\mathcal{S}_i$ is binary and $\mathcal{H}$ is post-unique, there can be no such action in A. Hence, *Rfind* will return *nil* and the algorithm will fail. □

Lemma 4.33 If there is a $k > 0$ s.t. P_{k+1} does not exist, then the algorithm fails in the *k+1*:st iteration of the loop in line 21–39. □

Proof: If P_{k+1} does not exist, then there is an $a \in P_k$ s.t. $f(a)[i] \not\sqsubseteq s_o[i]$, there is no $a' \in T_k$ s.t. $e(a')[i] = f(a)[i]$ and there is either no $a_1 \in A_k$ s.t $b(a_1)[i] = s_o[i]$ or no $a_2 \in A_k$ s.t. $b(a_2)[i] = f(a)[i]$. Since $\mathcal{S}_i$ is binary, $\mathcal{H}$ is post-unique and $\mathcal{A}$ contains at most one action of each type, we have $a' = a_1$ and $a_2 = \overline{a_1}$. We know that $P = P_k$ during the double **for** loop, so $a \in P$ during the whole loop. Either $a_1 \notin A_k$ or $a_2 \notin A_k$. Suppose $a_1 \notin A_k$, then $a_1 \notin \mathcal{A}$ since $a' = a_1$, $a' \notin T_k$ and, from definition 4.5, $\mathcal{A} = T_k \cup A_k$. Because $a \in P$, *Find* will be called to search D for a', but, since $a' \notin \mathcal{A}$ and $D \subseteq \mathcal{A}$, $a' \notin D$ so the search will fail. Consequently, *Rfind* will be called to search A for a_1, but this search will fail since $a_1 = a'$ and $A \subseteq \mathcal{A}$, so the algorithm will fail. Now suppose that $a_2 \notin A_k$. We know $a' \notin D$ so *Rfind* will search A for a_1 and a_2 during some iteration of the double **for** loop, but the search for a_2 will fail, and so will the algorithm. □

Lemma 4.34 If Δ does not exists, then the algorithm fails before line 40. □

Proof: If Δ does not exist, then there is a $k \geq 0$ s.t. P_k does not exist, so, by lemmata 4.32 and 4.33, the algorithm will fail before line 41. □

Lemma 4.35 If Δ exists, then $r = \delta$ after line 52. □

Proof: By lemma 4.34 the algorithm will go through lines 41–53 if Δ exists. First, r is initialized as a $|D| \times |D|$ zero matrix. For each pair a, a' of actions in D, the a, a' entry in r is marked 1 if either $e(a)[i] = f(a')[i]$ or $b(a')[i] = f(a)[i]$, corresponding to $a\gamma a'$ and $a\eta a'$ respectively. Hence r is a relation matrix for θ_D in line 51, and in line 52 r is set to the transitive closure of itself, thus yielding the transitive closure of θ_D, i.e. δ_D. By lemma 4.31, $D = \Delta$ after line 39, so $r{=}\delta_D = \delta$. □

Theorem 4.36 Algorithm 4.1 returns a parallel minimal plan from s_o to $s_\star$ if there is any plan from s_o to $s_\star$ and otherwise it fails. $\Box$

Proof: Straightforward from lemmata 4.31, 4.34, 4.35 and theorems 4.24 and 4.25. $\Box$

4.4 Complexity Results for SAS-PUBS Planning

This subsection is devoted to the complexity analysis of algorithm 4.1. The first result is theorem 4.38 stating the time complexity of the algorithm is polynomial in the number of features. We also analyse the complexity of deciding whether a given problem is in the SAS-PUBS class and theorem 4.41 states that the total complexity of both finding whether the algorithm is applicable and, if so, apply it is also polynomial in the number of features. Finally, the space complexity is stated in theorem 4.42. Our goal is only to prove that the algorithm is tractable, so no attempts have been made to reduce the complexity figure further. We follow the notation used by Baase (1988).

Lemma 4.37 $O(|\mathcal{H}|) \subseteq O(|\mathcal{M}|)$ for SAS-PUBS problems. $\Box$

Proof: $\mathcal{H}$ is post-unique, so $\mathcal{H}$ contains at most $|\mathcal{S}_i|$ action types affecting i for each $i \in \mathcal{M}$. Since $\mathcal{S}_i$ is binary for all $i \in \mathcal{M}$, $|\mathcal{H}| \leq 2|\mathcal{M}|$, from which the lemma follows trivially. $\Box$

Theorem 4.38 Algorithm 4.1 runs in $O(|\mathcal{M}|^3)$ time, worst-case. $\Box$

Proof: As basic operations we take variable assignment, elementary pointer operations, and comparison of two feature values, all of which are constant time operations. For simplicity, we assume that states are represented as arrays and sets as unordered linked lists. Consequently, the number of operations used by *Find* and *Rfind* is linear in the size of the set searched and the number of operations used by *Insert* is constant.

1. Initializing D and P takes a constant number of operations.

2. The **for** loop in line 12–19 does $|\mathcal{M}|$ iterations and, in the worst case, the loop body searches A for an action, which takes $O(|A|)$ operations. $A \subseteq \mathcal{A}$ and, by def 4.5, $|\mathcal{A}| = |\mathcal{H}|$, giving $|A| \leq |\mathcal{A}| = |\mathcal{H}|$, so the search takes $O(|A|) \subseteq O(|\mathcal{H}|) \subseteq O(|\mathcal{M}|)$ operations. Hence, the whole loop does $O(|\mathcal{M}|^2)$ operations in the worst case.

3. To analyze the **while** loop in line 21–39, we must first determine how many iterations are done by the **while** loop and the outer **for** loop (iterating over P). Let m be the smallest k s.t. $P_k = \varnothing$. We observe that the **while** loop terminates as soon as $P = \varnothing$, and, by lemmata 4.29 and 4.30, $P = P_k$ after the k:th iteration of the loop, so the loop terminates after th m:th iteration. By lemma 4.28, $\cup_{k=0}^{m} P_k = \Delta \subseteq \mathcal{A}$, so, since all P_k are disjoint, the body of the combined **while** loop and outer **for** loop does $\sum_{k=0}^{m} |P_k| = |\Delta| \leq |\mathcal{A}| = |\mathcal{H}|$ iterations. The inner **for** loop does $|\mathcal{M}|$ turns, so the body of the inner **for** loop is executed $O(|\mathcal{H}||\mathcal{M}|)$ times. In worst case, the loop body searches D once and A twice, but $D \subseteq \mathcal{A}$ and $A \subseteq \mathcal{A}$ so the loop body does $O(|\mathcal{A}|) = O(|\mathcal{H}|)$ operations. Hence, the **while** loop does $O(|\mathcal{H}|^2|\mathcal{M}|) \subseteq O(|\mathcal{M}|^3)$ operations in the worst case.

4. Initializing r takes $\Theta(|D|^2)$ operations, but $D \subseteq \mathcal{A}$, so $\Theta(|D|^2) \subseteq O(|\mathcal{A}|^2) = O(|\mathcal{H}|^2) \subseteq O(|\mathcal{M}|^2)$. Hence, the initialization takes $O(|\mathcal{M}|^2)$ operations.

5. The double **for** loop in line 43–50 does $\Theta(|D|^2|\mathcal{M}|) \subseteq O(|\mathcal{M}|^3)$ operations.

6. Baase (1988) proves that Warshalls algorithm can be used to compute the transitive closure of any relation over D using $\Theta(|D|^3) \subseteq O(|\mathcal{M}|^3)$ operations.

Clearly, the algorithm does $O(|\mathcal{M}|^3)$ operations in the worst case. □

An alternative 'standard' method for SAS-PUBS planning would be to construct a graph with the states in $\mathcal{S}$ as vertices and the actions in $\mathcal{A}$ as arcs. Assuming that all arcs have unit cost, we could apply a shortest path algorithm to the graph. Unfortunately, the time complexity for constructing the graph is exponential in $|\mathcal{M}|$ and the time consumed by the shortest path algorithm is at least linear in the size of the graph , so this approach is no effective alternative to our algorithm. On the other hand, this latter method is applicable to a larger class of problems than the SAS-PUBS class.

It is also appropriate to remark on the parameter used for measuring time complexity. While the number of actions in the solution, that is the generated plan, is usually used as parameter, for example by Chapman (1987) and Dean and Boddy

(1988), we use the number of features. We find our parameter more natural since it is known in advance, while, in the former approach, we first have to find the solution before we can say how hard it is to find. In defense of the former approach, it should be said that, for constraint-posting planners with infinite domains, it is hard to measure the complexity in other parameters.

Theorem 4.39 Deciding whether a given SAS problem is in the SAS-PUBS class can be done in $O(|\mathcal{M}|^3)$ time. □

Proof: Testing whether $\mathcal{S}$ is binary requires examining $\mathcal{S}_i$, for each $i \in \mathcal{M}$, to see whether it contains more than two elements or not. This requires $O(|\mathcal{M}|)$ operations. Testing whether $\mathcal{H}$ is unary requires testing for each $h \in \mathcal{H}$ whether there is more than one $i \in \mathcal{M}$ s.t. $b(h)[i] \neq u$. This can be done in $O(|\mathcal{H}||\mathcal{M}|) \subseteq O(|\mathcal{M}|^2)$ time. To test whether $\mathcal{H}$ is post-unique requires examining each pair $h, h' \in \mathcal{H}$ to see if $e(h)[i] = e(h')[i]$ for some $i \in \mathcal{M}$. This can be done in $O(|\mathcal{H}|^2|\mathcal{M}|) \subseteq O(|\mathcal{M}|^3)$ time. Checking whether $\mathcal{H}$ is single-valued requires checking for each pair $h, h' \in \mathcal{H}$ whether $u \neq f(h)[i] \neq f(h')[i] \neq u$ for some $i \in \mathcal{M}$. This takes $O(|\mathcal{H}|^2|\mathcal{M}|) \subseteq O(|\mathcal{M}|^3)$ time. Consequently, deciding whether a given SAS problem is SAS-PUBS thus takes $O(|\mathcal{M}|^3)$ time. □

Theorem 4.40 Testing whether $\mathcal{H}$ fulfills axioms 2.3–2.7 takes $O(|\mathcal{M}|^3)$ time. □

Proof: Testing whether $\mathcal{H}$ fulfills the first four axioms requires looping through $\mathcal{H}$ and $\mathcal{M}$ and thus takes $O(|\mathcal{H}||\mathcal{M}|) \subseteq O(|\mathcal{M}|^2)$ time. Testing that the last axiom is fulfilled requires testing each $i \in \mathcal{M}$ for each pair $h, h' \in \mathcal{H}$ and thus takes $O(|\mathcal{H}|^2|\mathcal{M}|) \subseteq O(|\mathcal{M}|^3)$ time. Hence, testing whether $\mathcal{H}$ fulfills the axioms takes $O(|\mathcal{M}|^3)$ time. □

Theorem 4.41 Given a planning problem $\langle \mathcal{M}, \mathcal{S}_1, \ldots, \mathcal{S}_{|\mathcal{M}|}, \mathcal{H}, s_o, s_\star \rangle$ fulfilling definitions 2.1 and 2.2, it takes $O(|\mathcal{M}|^3)$ time to decide whether algorithm 4.1 is applicable and, if so, find a minimal plan from s_o to $s_\star$ or report that no plan at all exists from s_o to $s_\star$. □

Proof: Immediate from theorems 4.38, 4.39 and 4.40. □

Theorem 4.42 Algorithm 4.1 uses $O(|\mathcal{M}|^2)$ space. □

Proof: We assume that states are represented as arrays of feature values. We further assume that actions are represented as tuples $\langle l, b(h), f(h) \rangle$, that is, we represent the action type by its corresponding pre- and prevail-conditions. Note that the post-condition is implicit in the pre-condition for the SAS-PUBS class. Sets are assumed to be represented as linked lists, and matrices as arrays.

State and action variables clearly use $O(|\mathcal{M}|)$ space. $P \subseteq A$, $Q \subseteq A$, $D \subseteq A$ and $A \subseteq \mathcal{A}$, so P, D, Q, and A contain $O(|\mathcal{A}|) = O(|\mathcal{H}|) \subseteq O(|\mathcal{M}|)$ actions. Hence, each of these variables occupy $O(|\mathcal{M}|^2)$ space. The relation matrix r is of size $O(|D|^2) \subseteq O(|\mathcal{A}|^2) \subseteq O(|\mathcal{M}|^2)$. The total space required by the algorithm is clearly $O(|\mathcal{M}|^2)$. □

5 Example

In this section we apply our planning algorithm to a simple example. The problem is to refuel an aircraft using a mobile refuel vehicle, an example chosen for pedagogical reasons rather than realism. We define four features such that for any state $s \in \mathcal{S}$, the state $s = \langle s[1], s[2], s[3], s[4] \rangle$ is interpreted as:

$$
\begin{aligned}
s[1] &= \begin{cases} 0 \text{ if the tank of the aircraft is empty} \\ 1 \text{ if the tank of the aircraft is full} \end{cases} \\
s[2] &= \begin{cases} 0 \text{ if the refuel vehicle is not at the aircraft} \\ 1 \text{ if the refuel vehicle is at the aircraft} \end{cases} \\
s[3] &= \begin{cases} 0 \text{ if the aircraft is not grounded} \\ 1 \text{ if the aircraft is grounded} \end{cases} \\
s[4] &= \begin{cases} 0 \text{ if the tank of the aircraft is open} \\ 1 \text{ if the tank of the aircraft is not open} \end{cases}
\end{aligned}
$$

When refuelling the aircraft, it is important to eliminate the voltage difference between the aircraft and the refuel vehicle in order to avoid sparks. That the aircraft is grounded thus means that such an electrical connection is established, so grounding has nothing to do with whether the aircraft is airborne or not. There are seven action types in $\mathcal{H}$, and these are defined together with their pre-, post-, and prevail-conditions in table 2. We furthermore assume $\mathcal{L}$ to consist of the natural numbers and we let

$$\mathcal{A} = \{\langle 1, refuel \rangle, \langle 2, vehicle\_to\_aircraft \rangle, \langle 3, vehicle\_from\_aircraft \rangle,$$

$$\langle 4, ground\rangle, \langle 5, unground\rangle, \langle 6, close\_tank\rangle,$$
$$\langle 7, open\_tank\rangle\}.$$

The initial state is

$$s_o = \langle 0,0,0,1\rangle$$

and the final state is

$$s_\star = \langle 1,0,0,1\rangle$$

which means that we want to refuel the aircraft. The problem of finding a plan from s_o to $s_\star$ is clearly in the SAS-PUBS class. We will now work through the algorithm on this example.

The for loop in lines 12–19 test for every $i \in \mathcal{M}$ whether $s_o[i] \neq s_\star[i]$ and, if this is the case, *Rfind* is called to search A for an action that changes the i:th feature from $s_o[i]$ to $s_\star[i]$. If such an action is found *Rfind* removes it from A and inserts it into D and P. Now, $s_o[1] \neq s_\star[1]$ so *Rfind* searches A for an action a s.t. $b(a)[1] = s_o[1]$ and thus implicitly also $e(a)[1] = s_\star[1]$. The only action in A satisfying this condition is $\langle 1, refuel\rangle$ which is deleted from A by *Rfind* and inserted into D and P. The states s_o and $s_\star$ are equal for all other features so we have the following variable values after the for loop

$$\begin{aligned}
A &= \{\langle 2, vehicle_to_aircraft\rangle, \langle 3, vehicle_from_aircraft\rangle, \langle 4, ground\rangle,\\
&\quad \langle 5, unground\rangle, \langle 6, close_tank\rangle, \langle 7, open_tank\rangle\}\\
D &= \{\langle 1, refuel\rangle\}\\
P &= \{\langle 1, refuel\rangle\}
\end{aligned}$$

Since $P \neq \varnothing$ we go through the while loop in lines 21–39 and each $a \in P$ is processed by the for loop in lines 24–36. Without loss of generality we assume the actions in P are processed in the order they were inserted into P.

The first time through the while loop there is only one action, $\langle 1, refuel\rangle$ in P. The inner for loop goes through all $i \in \mathcal{M}$ and test whether $f(\langle 1, refuel\rangle)[i] \not\sqsubseteq s_o[i]$. If this is the case, the algorithms tries to satisfy this condition either by finding an set action for i already in D or by inserting a set/reset pair for i into D. $f(\langle 1, refuel\rangle)[1] = u$ so nothing happens the first time through the loop. However, $f(\langle 1, refuel\rangle)[2] = 1 \not\sqsubseteq s_o[2] = 0$ so *Find* is called to search D for an action a s.t. $b(a)[2] = 1$. No such a exists in D so the search fails and *Rfind* is instead called to search A for a set/reset pair a_1, a_2 for feature 2. This succeeds with $a_1 = \langle 2, vehicle\_to\_aircraft\rangle$ and $a_2 = \langle 3, vehicle\_from\_aircraft\rangle$ which are removed from A and inserted into D and Q. We now have

$$A = \{\langle 4, ground\rangle, \langle 5, unground\rangle, \langle 6, close\_tank\rangle, \langle 7, open\_tank\rangle\}$$

$$\begin{aligned}
D &= \{\langle 1, refuel\rangle, \langle 2, vehicle_to_aircraft\rangle, \langle 3, vehicle_from_aircraft\rangle\} \\
P &= \{\langle 1, refuel\rangle\} \\
Q &= \{\langle 2, vehicle_to_aircraft\rangle, \langle 3, vehicle_from_aircraft\rangle\}
\end{aligned}$$

The same procedure is repeated for features 3 and 4 and the set/reset pairs $\langle 4, ground\rangle, \langle 5, unground\rangle$ and $\langle 6, close\_tank\rangle, \langle 7, open\_tank\rangle$ respectively are found. Immediately before line 38 we have the following variable values

$$\begin{aligned}
A &= \varnothing \\
D &= \{\langle 1, refuel\rangle, \langle 2, vehicle_to_aircraft\rangle, \langle 3, vehicle_from_aircraft\rangle, \\
&\quad \langle 4, ground\rangle, \langle 5, unground\rangle, \langle 6, close_tank\rangle, \langle 7, open_tank\rangle\} \\
P &= \{\langle 1, refuel\rangle\} \\
Q &= \{\langle 2, vehicle_to_aircraft\rangle, \langle 3, vehicle_from_aircraft\rangle, \langle 4, ground\rangle, \\
&\quad \langle 5, unground\rangle, \langle 6, close_tank\rangle, \langle 7, open_tank\rangle\}
\end{aligned}$$

and P is then set to Q so

$$\begin{aligned}
P &= \{\langle 2, vehicle_to_aircraft\rangle, \langle 3, vehicle_from_aircraft\rangle, \langle 4, ground\rangle \\
&\quad \langle 5, unground\rangle, \langle 6, close_tank\rangle, \langle 7, open_tank\rangle\}
\end{aligned}$$

and we go through the while loop a second time. The first two actions removed from P are $\langle 2, vehicle\_to\_aircraft\rangle$ and $\langle 3, vehicle\_from\_aircraft\rangle$, but $f(\langle 2, vehicle\_to\_aircraft\rangle)[i] = f(\langle 3, vehicle\_from\_aircraft\rangle)[i] = u$ for all $i \in \mathcal{M}$ so A, D, P and Q remain invariant the first two times through the outer for loop. The third action removed from P is $\langle 4, ground\rangle$ and $f(\langle 4, ground\rangle)[i] = u$ for all $i \in \mathcal{M}$ except for $i = 2$. However, *Find* searches D for an action a' in D s.t. $b(a')[2] = s_o[2]$ and succeeds with $a' = \langle 2, vehicle\_to\_aircraft\rangle$. Since *Find* succeeds, nothing more happens this time through the loop and A, D, P and Q remain invariant. The remaining three actions in P are handled the same way so immediately before line 38 we have

$$\begin{aligned}
A &= \varnothing \\
D &= \{\langle 1, refuel\rangle, \langle 2, vehicle_to_aircraft\rangle, \langle 3, vehicle_from_aircraft\rangle, \\
&\quad \langle 4, ground\rangle, \langle 5, unground\rangle, \langle 6, close_tank\rangle, \langle 7, open_tank\rangle\} \\
P &= \{\langle 2, vehicle_to_aircraft\rangle, \langle 3, vehicle_from_aircraft\rangle, \langle 4, ground\rangle, \\
&\quad \langle 5, unground\rangle, \langle 6, close_tank\rangle, \langle 7, open_tank\rangle\} \\
Q &= \varnothing
\end{aligned}$$

and since P is set to $Q = \varnothing$ the while loop terminates.

The variable r is now initialized to a zero matrix of sufficient size to hold a relation over D. The outer two for loops in lines 43–50 go through all pairs a, a' of actions in D. For each such pair, the inner for loop records in r that a is related to a' if there is some $i \in \mathcal{M}$ s.t. $a\gamma a'$ or $a\eta a'$ according to definition 4.6. Finally, r is set to be the transitive closure of itself. The algorithm returns the plan $\langle D, r\rangle$ where

$$\begin{aligned} D \;=\; &\{\langle 1, refuel\rangle, \langle 2, vehicle\_to\_aircraft\rangle, \langle 3, vehicle\_from\_aircraft\rangle, \\ &\langle 4, ground\rangle, \langle 5, unground\rangle, \langle 6, close\_tank\rangle, \langle 7, open\_tank\rangle\} \end{aligned}$$

and the relation r is depicted in figure 3 (with transitive arcs omitted).

6 Discussion of the SAS-PUBS Class

In this section, we will briefly discuss the restrictions of the SAS-PUBS class.

The restriction to binary domains is fairly harmless if $\mathcal{H}$ is not restricted to be unary, since a non-binary domain $\mathcal{S}_i$ could be represented by $\lceil \log_2 |\mathcal{S}_i| \rceil$ binary domains, if the axioms are modified accordingly. Expressibility is retained, but the time complexity increases since the number of domains, $|\mathcal{M}|$, increases. On the other hand, when $\mathcal{H}$ is unary, as in the SAS-PUBS class, it is obviously a serious restriction to require binary domains; many planning problems require multi-valued features. Of course, this restriction appears in all planners using propositional logic for state modelling. 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 multi-valued features does not help much in the general case. Although one could represent several feature domains as one multi-valued feature, this would most likely lead to violations of axiom 2.6 for most planning problems. 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. Nevertheless, 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 another action type requires the same valve to be closed. Prevail-minimality has no effect on the expressibility, but time complexity is increased if $\mathcal{H}$ is not prevail minimal.

Since single-valuedness seems to be the most serious restriction, it would be natural to try to eliminate that restriction first. This would result in the SAS-PUB class (Post-unique, Unary and Binary). The SAS-PUB class is, although very restricted, believed to be applicable to some realistic problems in sequential control. Unfortunately, the worst-case time for SAS-PUB planning is exponential in the number of features.

Theorem 6.1 A lower bound for SAS-PUB planning is $\Omega(2^{|\mathcal{M}|})$ operations in the worst case. □

Proof: We first note that a plan is not minimal if it passes some state $s \in \mathcal{S}$ more than once. Since there are $2^{|\mathcal{M}|}$ states, no minimal plan can have more than $2^{|\mathcal{M}|} - 1$ actions. We will now prove that there are SAS-PUB problems with minimal plans of this size by constructing a generic example. Given an integer $m > 0$, let $\mathcal{M} = \{1, 2, \ldots, m\}$ and let $\mathcal{S}_i = \{0, 1\}$ for $i \in \mathcal{M}$. Construct $\mathcal{H} = \{h_1, h'_1, \ldots, h_m, h'_m\}$ as follows:

For all k

$$b(h_k)[i] = e(h'_k)[i] = \begin{cases} 0, & i = k \\ u, & i \neq k \end{cases}$$

$$e(h_k)[i] = b(h'_k)[i] = \begin{cases} 1, & i = k \\ u, & i \neq k \end{cases}$$

$$f(h_k)[i] = f(h'_k)[i] = \begin{cases} 0, & 1 \leq i < k-1 \\ 1, & 1 \leq i = k-1 \\ u, & k \leq i \leq m \end{cases}$$

Also define s_o and $s_\star$ s.t.

$$s_o[i] = 0 \text{ for } 1 \leq i \leq m$$

$$s_\star[i] = \begin{cases} 0, & 1 \leq i < m \\ 1, & i = m \end{cases}$$

It can be proven by induction on m that a minimal plan from s_o to $s_\star$ requires $2^m - 1$ actions.

Obviously, minimal plans for SAS-PUB problems are of size $\Theta(2^{|\mathcal{M}|})$ in the worst case, so a trivial lower bound for worst case planning is $\Omega(2^{|\mathcal{M}|})$ operations.

□

It is thus not possible to construct a polynomial-time planning algorithm for SAS-PUB problems. However, we conjecture that it is possible to replace single-valuedness with other restrictions that are fulfilled for many (or even most) practical SAS-PUB problems, but which reduces the complexity drastically.

7 Discussion

We have identified a class of sequential deterministic planning problems, the SAS-PUBS class, and presented an algorithm for finding minimal plans in this class. The algorithm is proven sound and complete and runs in polynomial time. This result provides a kind of lower bound for planning; at least this class of problems can be solved in polynomial time. The SAS-PUBS class is thus of theoretical interest, even if it is of limited practical interest.

The class one gets when lifting the single-valuedness restriction is, however, far more 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 within these areas 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. An ordinary paper manufacturing plant can have 10000 to 15000 sensors, so the number of state variables could be well in excess of that figure.

We have shown that the worst case lower bound for planning in this extended class is of $\Omega(2^{|\mathcal{M}|})$ time, but this figure corresponds to theoretical cases that seem almost pathological for any practical application. It arises when the minimal plans themselves are exponentially sized and no one is likely to want such a plan since it would take to long time to execute for anything but trivially small problems. We have some ideas about how to replace single-valuedness with other restrictions that, hopefully, reduces the complexity figure without severely restricting the practical usefulness. It is also worth noticing that the SAS-PUBS planning algorithm described in the previous section is sound even if the set of actions is not single-valued.

One interesting branch of future research is thus to investigate how the different restrictions affects complexity of SAS planning problems. Another interesting branch would be to, perhaps, retain some of the restrictions but allow extended actions structures with keep-conditions (Bäckström, 1988a; Bäckström, 1988b) or interval-valued features (Bäckström, 1988b; Bäckström, 1988c). It would also be interesting to relate the work in this article to other planning formalisms and see whether other planners enjoy the same complexity results under the same restrictions as we use.

The planning algorithm has been implemented in C on a Sun[3] SPARCstation 1 and runs fairly fast although not optimized. An earlier version of the algorithm has been implemented in prolog and extended with heuristics to handle a small subclass of SAS-PUB problems (Klein, 1990). This implementation has also been augmented with some heuristics for dealing with slightly larger classes. This has worked well for some test examples, but no theoretical results exist regarding this augmentation. A system for executing parallel plans expressed in action structures has been implemented by Hultman (1987a; 1987b; 1988).

Acknowledgments

This research was supported by the Swedish Board of Technical Development. The authors would also like thank the anonymous reviewers for their comments on how to improve this article.

[3]Sun and SPARCstation 1 are trademarks of Sun Microsystems, Inc.

References

Allen, J. F. (1981). A general model of action and time. Technical Report 97, Computer Science Department, University of Rochester, NY.

Allen, J. F. (1984). Towards a general theory of action and time. *Artificial Intelligence*, 23:123–154.

Baase, S. (1988). *Computer Algorithms: Introduction and Analysis*. Addison Wesley, Reading, Mass. 2nd edition.

Bäckström, C. (1988a). 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.

Bäckström, C. (1988b). Reasoning about interdependent actions. Licentiate Thesis 139, Department of Computer and Information Science, Linköping University, Linköping, Sweden.

Bäckström, C. (1988c). 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.

Brachman, R. J. and Levesque, H. J. (1984). 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.

Brown, F., editor (1987). *The Frame Problem in Artificial Intelligence, Proceedings of the 1987 Workshop*, Lawrence, Kansas.

Chapman, D. (1987). Planning for conjunctive goals. *Artificial Intelligence*, 32:333–377.

Dean, T. and Boddy, M. (1988). Reasoning about partially ordered events. *Artificial Intelligence*, 36:375–399.

Fikes, R. E. and Nilsson, N. J. (1971). Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208.

Hansson, C. (1990). A prototype system for logical reasoning about time and action. Licentiate Thesis 203, Department of Computer and Information Science, Linköping University, Linköping, Sweden.

Hayes, P. J. (1981). The frame problem and related problems in artificial intelligence. In Webber, B. L. and Nilsson, N. J., editors, *Readings in Artificial Intelligence*, pages 223–230. Morgan Kaufman, Los Altos, Ca.

Hultman, J. (1987a). COPPS - A software system for defining and controlling actions in a mechanical system. In *IEEE Workshop on Languages and Automation*, Vienna, Austria.

Hultman, J. (1987b). COPPS - A software system for defining and controlling actions in a mechanical system. Research Report LiTH-IDA-R-87-06, Department of Computer and Information Science, Linköping University, Linköping, Sweden.

Hultman, J. (1988). A software system for defining and controlling actions in a mechanical system. Licentiate thesis 146, Department of Computer and Information Science, Linköping University, Linköping, Sweden.

Klein, I. (1990). Planning for a class of sequential control problems. Licentiate Thesis 234, Department of Electrical Engineering, Linköping University, Linköping, Sweden.

Levesque, H. J. and Brachman, R. J. (1985). A fundamental tradeoff in knowledge representation and reasoning (revised version). In Brachman, R. J. and Levesque, H. J., editors, *Readings in Knowledge Representation*, pages 41–70. Morgan Kaufman, Los Altos, Ca.

Mendelson, E. (1987). *Introduction to Mathematical Logic*. Wadsworth & Brooks, Monterey, Ca, third edition.

Newell, A. and Simon, H. A. (1972). *Human Problem Solving*. Prentice Hall, Englewood Cliffs, NJ.

Sandewall, E. (1988a). Formal semantics for reasoning about change with ramified causal minimizations. Research Report LiTH-IDA-88-08, Department of Computer and Information Science, Linköping University, Linköping, Sweden.

Sandewall, E. (1988b). Non-monotonic entailment for reasoning about time and action. Part I: Sequential actions. Research Report LiTH-IDA-R-88-27, Department of Computer and Information Science, Linköping University, Linköping, Sweden.

Sandewall, E. (1988c). Non-monotonic entailment for reasoning about time and action. Part II: Concurrent actions. Research Report LiTH-IDA-R-88-28, Department of Computer and Information Science, Linköping University, Linköping, Sweden.

Sandewall, E. (1988d). Non-monotonic entailment for reasoning about time and action. Part III: Decision procedure. Research Report LiTH-IDA-R-88-29, Department of Computer and Information Science, Linköping University, Linköping, Sweden.

Sandewall, E. (1989). A decision procedure for a theory of actions and plans. In *Proceedings of the Fourth International Symposium on Methodologies for Intelligent systems (ISMIS-89)*, pages 501–514, Charlotte, NC.

Sandewall, E. and Rönnquist, R. (1986a). A representation of action structures. In *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI-86)*, pages 89–97, Philadelphia, Pennsylvania.

Sandewall, E. and Rönnquist, R. (1986b). A representation of action structures. Research Report LiTH-IDA-R-86-13, Department of Computer and Information Science, Linköping University, Linköping, Sweden.

Shoham, Y. (1987). Temporal logics in AI: Semantical and ontological considerations. *Artificial Intelligence*, 33:89–104.

Wilkins, D. E. (1988). *Practical Planning*. Morgan Kaufman, San Mateo, Ca.

Figure Captions

Fig. 1. The lattice $\langle \mathcal{S}_1^+, \sqsubseteq \rangle$ in example 2.1.

Fig. 2. The lattice $\langle \mathcal{S}^+, \sqsubseteq \rangle$ in example 2.1.

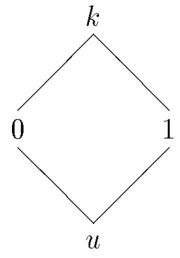
Fig. 3. The relation δ_Δ on the set $\Delta(s_o, s_\star)$ in the refuelling example. The transitive arcs are omitted.

h	$b(h)$	$e(h)$	$f(h)$
h_1	$\langle u, u \rangle$	$\langle u, 1 \rangle$	$\langle k, u \rangle$
h_2	$\langle u, 0 \rangle$	$\langle u, 0 \rangle$	$\langle 1, 0 \rangle$
h_3	$\langle 0, u \rangle$	$\langle 1, u \rangle$	$\langle u, 1 \rangle$
h_4	$\langle 0, u \rangle$	$\langle 1, u \rangle$	$\langle u, 1 \rangle$

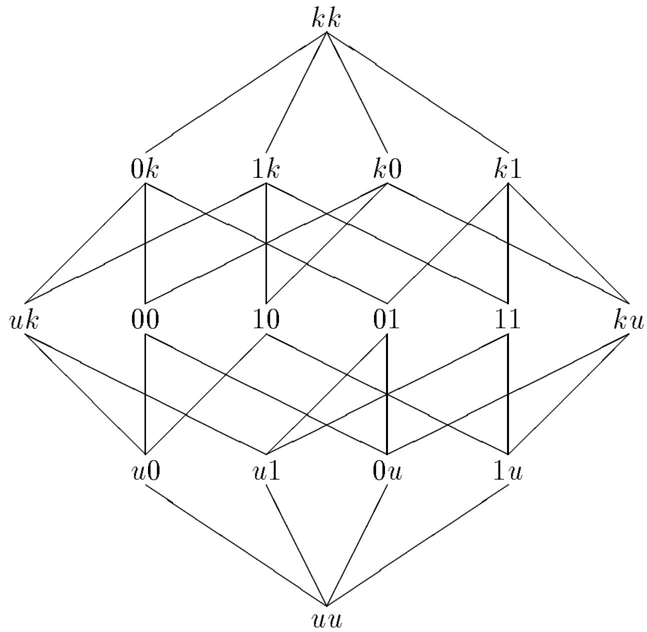
Table 1. Action types for example 2.2.

action type (h)	*b(h)*	*e(h)*	*f(h)*
refuel	$\langle 0, u, u, u \rangle$	$\langle 1, u, u, u \rangle$	$\langle u, 1, 1, 0 \rangle$
move_vehicle_to_aircraft	$\langle u, 0, u, u \rangle$	$\langle u, 1, u, u \rangle$	$\langle u, u, u, u \rangle$
move_vehicle_from_aircraft	$\langle u, 1, u, u \rangle$	$\langle u, 0, u, u \rangle$	$\langle u, u, u, u \rangle$
ground	$\langle u, u, 0, u \rangle$	$\langle u, u, 1, u \rangle$	$\langle u, 1, u, u \rangle$
unground	$\langle u, u, 1, u \rangle$	$\langle u, u, 0, u \rangle$	$\langle u, 1, u, u \rangle$
close_aircraft_tank	$\langle u, u, u, 0 \rangle$	$\langle u, u, u, 1 \rangle$	$\langle u, 1, u, u \rangle$
open_aircraft_tank	$\langle u, u, u, 1 \rangle$	$\langle u, u, u, 0 \rangle$	$\langle u, 1, u, u \rangle$

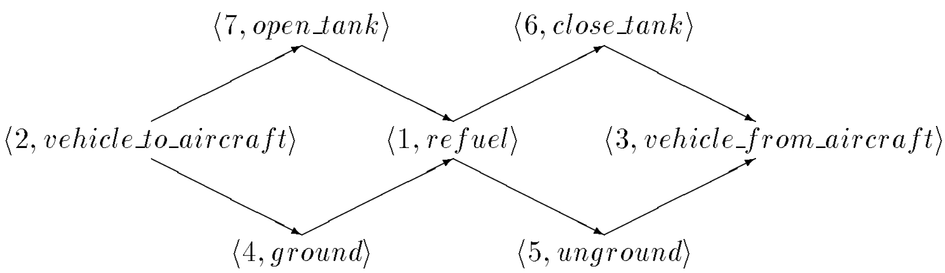
Table 2. Action types for the aircraft example.



Bäckström, Klein: Planning in polynomial time: The SAS-PUBS class. Fig. 1.



Bäckström, Klein: Planning in polynomial time: The SAS-PUBS class. Fig. 2.



Bäckström, Klein: Planning in polynomial time: The SAS-PUBS class. Fig. 3.