Efficient planning for a miniature assembly line

Inger Klein\textsuperscript{a,}\textsuperscript{*}, Peter Jonsson\textsuperscript{b}, Christer Bäckström\textsuperscript{b}

\begin{itemize}
  \item \textsuperscript{a}Department of Electric Engineering, Linköping University, S-581 83 Linköping, Sweden
  \item \textsuperscript{b}Department of Computer and Information Science, Linköping University, S-581 83 Linköping, Sweden
\end{itemize}

Received 10 May 1997

Abstract

This paper presents a provably correct and efficient, polynomial time, planning tool and its application to a miniature assembly line for toy cars. Although somewhat limited, this process has many similarities with real industrial processes. One of our previous polynomial-time planning algorithms has been extended and adapted to work for a larger class of planning problems, including this particular process. The plans produced by the planner are then translated into GRAFCET charts, which are compiled into code for a programmable logic controller. Although capable of producing ordinary assembly plans, the system is mainly intended for producing plans in error recovery situations.

\textcopyright{} 1998 Elsevier Science Ltd. All rights reserved.

Keywords: AI planning; Sequential control; Assembly planning; Error recovery; Polynomial-time planning

1. Introduction and background

Artificial-intelligence action planning in its general form is known to be very hard. Theoretical analyses in the literature tell us that planning in the languages used by the well-known planners STRIPS [1] and TWEAK [2] is undecidable [2, 3]. Even when restricted to the propositional case, planning is still PSPACE-complete in these languages [4]. A number of sublanguages defined by syntactic restrictions have also been studied in the literature [3–5], most of these being computationally difficult as well.

However, complexity results of this type only tell us how difficult it is to plan for the hardest problems that can be encoded in a particular language and they do not reveal much about the inherent complexity of particular application problems. For instance, the ubiquitous blocks-world problem in its standard form can be encoded in the propositional STRIPS formalism. This tells us that the problem lies in the complexity class PSPACE, but it does not imply that the blocks-world problem is as hard as PSPACE-complete. In fact, it is known that finding an optimal plan for the blocks-world problem is only NP-complete and finding any plan is even computationally tractable [6]. In addition, the variant of this problem where several blocks can have identical labels seems not even to have any straightforward encoding in the propositional STRIPS language, yet this problem is still only NP-complete [7] (which tells us that there must actually exist some encoding, although we get no clue what it may look like). This type of phenomenon is likely to account for many of the discrepancies between theoretical difficulty results and reported successful applications of AI planning.

While AI planners used in applications typically use expressive languages, where planning is PSPACE-complete or even worse, the planners are usually equipped with heuristics to make them perform well on the application problems at hand. However, unless the application problem is inherently easy, heuristics cannot guarantee good performance. It may, perhaps, improve the average-case, or even the worst-case, performance, but will likely be at the expense of incompleteness, i.e. risking not finding a solution, even when there is one. Computational intractability is, in a sense, also to be regarded as a type of incompleteness, since the planner may not deliver a solution within reasonable time. Moreover, even when the application problem is provably easy, it may not be easy to design heuristics that make a planner perform well. A heuristics-based planner may go on tour in an infinite search space when a more specialised planning algorithm would find a solution quickly.

Although incompleteness or good average-case behaviour may be acceptable in some applications, it is not acceptable in others. For instance, many problems in sequential control within automatic control can be viewed as planning problems of the second type, where correct
solutions must be found quickly. Automatic control has a long tradition of using mathematically well-founded methods with provable properties and researchers in this area see the lack of such theories as one of the major problems with AI planning [8]. Benveniste and Åström [9] presented a study of how computer software is used in large-scale control applications, such as the process industry and metro traffic networks. One of their findings was that the industry asks for more mathematical tools for modelling dynamic systems of a combinatorial nature. The use of formal methods was considered a way to improve formal guarantees and reduce the complexity of the resulting code. Another, related, finding was that expert systems are frequently used in large-scale control applications, but never in safety-critical parts of the systems. In the planning world, this would roughly translate to saying that heuristics are not sufficient for safety-critical applications. This means that we have to aim at more formal methods than heuristics and other expert-knowledge-based methods whenever possible.

While there are many types of industrial problems that fit within the category just described, this article will focus on manufacturing, or more precisely, assembly planning. Using automated planners to generate the plan for normal operations may not be so interesting, since this plan is rarely changed and a lot of time and effort may thus be spent on designing and optimizing this plan by human experts. However, automated planning is an interesting tool for error recovery. Since the process may end up in any one of a very large number of states after a break-down, it is not realistic to have recovery plans designed in advance for all such states. It is, thus, desirable to invoke a planner in the case of break-downs, which finds a plan that brings the process back into normal operation again. Such a plan must be correct, in order not to jam or even destroy the plant, and it must also be found within a reasonable time, since large-scale processes typically have very high standstill costs.

Other cases where plan generation is useful is when the initial state is not fully specified until the plan is to be executed or when a human operator controls the process using higher-level commands that cannot be directly executed in the plant. In this latter case, the operator may, for instance, request that a certain valve be opened, but opening this valve actually requires several low-level actions to be performed, e.g. first closing another valve and then starting a pump, before opening the valve as requested. A planner based on a model of the plan could automatically transform the high-level request into the corresponding sequence of low-level actions, which together bring about the desired effect. However, a common problem in process plants is to revise the control program when the plant is somehow modified, e.g. by adding a new device. A remedy to this problem is not to develop the control program by hand, but instead build a modular model of the plant and use a planner to develop the control program based on this model. A local change in the plant should then correspond to a local change in the model only, ideally in a single module, and the problem of revising the control program is left to the planner.

In all the application examples described above, there is a need for a planning algorithm that not only produces correct plans, according to the model, but which is also efficient, ideally running in low-order polynomial time. The complexity analyses tell us that no general-purpose such planner can exist, however. On the other hand, it is possible that we can find efficient planners for certain restricted planning problems of practical interest. We have run a long-term project, which is aimed at identifying such problems, looking especially at problem areas in sequential control. Starting with a small toy problem in sequential control, we identified a number of inherent restrictions of this problem which, taken together, result in computational tractability. We first defined a formal planning problem, the SAS+-PUBS problem, based on these restrictions, and then proved this problem tractable by devising a provably correct, polynomial-time algorithm for it [10, 11]. We then continued, working from the bottom, up, successively removing and/or modifying restrictions to get increasingly more expressive, yet still tractable, planning languages [5, 12–14]. Polynomial-time algorithms were also presented for all these languages. The progress of, and philosophy behind, this research strategy is presented in Bäckström [15].

In this article, we report on the adaptation and extension of one of our planning algorithms, using the problem of generating assembly plans for a laboratory assembly line for toy cars, the LEGO car factory [16] (LEGO is a trademark of the LEGO company). Since this process resembles real industrial processes in many respects, it has been a long-standing target application for our research into tractable planning. We show that, by exploiting the inherent structure of the problem, we can model it and use a polynomial-time planning algorithm to solve it. More precisely, we use a simple and provably correct modification of our previously presented algorithm for the SAS+-IAO problem [14].

The basic idea of the approach is to split the problem instance into two partitions, such that one of these is an instance of the SAS+-IAO class and its solution serves as a partial plan (or a plan skeleton), which can then be completed by solving a number of subproblems from the remaining partition. In addition, we present a planning tool which uses this algorithm to generate a plan and then converts this plan into a corresponding GRAFCET [17] chart. A commercial compiler is then used to convert this GRAFCET chart into a code that can be executed on the programmable logic controller (PLC), which controls the LEGO car factory. Note that neither our algorithm nor the planning tool is specifically tailored for this assembly line, but is a general-purpose device.

This article is a compiled and extended version of two earlier conference publications [18, 19], and is structured as
follows. In Section 2 we describe our planning formalism and in Section 3 we recapitulate a previously reported subclass of this formalism. The new planning algorithm is described in Section 4 and the planning tool is described in Section 5. The LEGO-car factory is described in Section 6 and its modelling in the SAS formalism is described in Section 7. Section 8 discusses how to apply the planner to the LEGO-car factory using this model. The paper ends with discussion and conclusions sections.

2. The SAS formalism

We use the SAS planning formalism [5, 14, 20], which can be viewed as a variation on the propositional version of the STRIPS formalism. The SAS formalism is, in fact, equivalent, under polynomial reduction, to most other common variants of propositional STRIPS [20]. Yet, the formalisms have different modelling properties, making them conceptually different. Some problems are more naturally expressed in one of the formalisms than in the other. For instance, in our experience, control engineers seem to find the SAS formalism much more appealing than the STRIPS formalism.

There are basically two major differences between propositional STRIPS and the SAS formalism. First, the propositional atoms in the STRIPS formalism, which can be viewed as two-valued state-variables, are generalised into state variables with an arbitrary number of values. Second, the operator pre-condition in the STRIPS formalism is split into two different conditions in the SAS formalism, the pre-condition and the prevail-condition respectively. The SAS pre-condition corresponds to that part of the STRIPS pre-condition which involves variables that get their values changed by the operator. The prevail-condition constitutes the remaining part of the STRIPS pre-condition, that is, those variables which are required to have a specific value in order to execute the operator, but where this value remains unaltered by the operator. The SAS formalism is defined as follows.

Definition 1. A SAS structure is a tuple, \( \Phi = (\mathcal{V}, \mathcal{O}) \), with components defined as follows:

- \( \mathcal{V} = \{v_1, \ldots, v_n\} \) is a set of state variables. Each variable \( v \in \mathcal{V} \) has an associated domain of values \( D_v \), which implicitly defines an extended domain \( D_v^+ = D_v \cup \{u\} \), where \( u \) denotes the undefined value. Further, the total state space \( \mathcal{S} = D_{v_1} \times \cdots \times D_{v_n} \) and the partial state space \( \mathcal{S}^+ = D_{v_1}^+ \times \cdots \times D_{v_n}^+ \) are implicitly defined. We write \( s[v] \) to denote the value of the variable, \( v \), in a state, \( s \).
- \( \mathcal{O} \) is a set of operators of the form \( (\text{pre}, \text{post}, \text{prv}) \), where \( \text{pre}, \text{post}, \text{prv} \in \mathcal{S}^+ \) denote the pre-, post- and prevail-conditions, respectively. \( \mathcal{O} \) is subject to the following two restrictions:

(R1) for all \( (\text{pre}, \text{post}, \text{prv}) \in \mathcal{O} \) and \( v \in \mathcal{V} \) if \( \text{pre}[v] \neq u \), then \( \text{pre}[v] \neq \text{post}[v] \neq u \),

(R2) for all \( (\text{pre}, \text{post}, \text{prv}) \in \mathcal{O} \) and \( v \in \mathcal{V} \), \( \text{post}[v] = u \) or \( \text{prv}[v] = u \).

An instance of the SAS planning problem is given by a quadruple \( I = < \mathcal{V}, \mathcal{O}, s_0, s^* > \) such that \( (\mathcal{V}, \mathcal{O}) \) is a SAS structure and \( s_0, s^* \in \mathcal{S}^+ \) denote the initial and goal state, respectively.

Restriction R1 essentially says that the value of a state variable can never be made undefined, once defined by some operator, and that an operator affecting a variable must actually change its value. Restriction R2 says that the post- and prevail-conditions of an operator must never define the same variable. We further write \( x \subseteq y \) if the variable value \( x \) is subsumed by the value \( y \), i.e. if \( x = u \) or \( x = y \). We extend this notion to states, defining:

\[ s \subseteq t \text{ iff } \forall v \in \mathcal{V} \text{ either } s[v] = u \text{ or } s[v] = t[v]. \]

Moreover, for a SAS operator \( o = (\text{pre}, \text{post}, \text{prv}) \), we write \( \text{pre}(o), \text{post}(o) \) and \( \text{prv}(o) \) to denote \( \text{pre}, \text{post} \) and \( \text{prv} \) respectively. Finally, the notion \( \text{Seqs}(\mathcal{O}) \) denotes the set of all sequences that can be formed using the members of an operator set \( \mathcal{O} \), including the empty sequence and sequences with multiple occurrences of the same operator. Any such sequence is called a SAS plan (or simply a plan) over \( \Phi \).

A plan is a solution to a planning-problem instance if it is executable in the initial state, and it leads to the goal state. In order to define this concept formally, it is useful to first define an update operator, \( \oplus \), used to modify a state, \( s \), as specified by another state, \( t \). The meaning of \( s \oplus t \) is thus, that, in the resulting state, any variable having a defined value in the second state, \( t \), will get this value, and all other variables will retain their values from the first state, \( s \).

Definition 2. Given two states \( s, t \in \mathcal{S}^+ \), we define for all variables \( v \in \mathcal{V} \):

\[ (s \oplus t)[v] = \begin{cases} t[v], & \text{if } t[v] \neq u, \\ s[v], & \text{otherwise}. \end{cases} \]

This construction is used to describe the effects of operators such that if an operator, \( o \), is successfully executed in a state, \( s \), then the resulting state will be the state, \( s \), updated by the post-condition of \( o \), that is, the resulting state is \( s \oplus \text{post}(o) \). Based on this, we can define the concept of a valid plan. An operator, \( o \), can be successfully executed in a state, \( s \), if both its pre-condition and its prevail-condition are subsumed by this state, that is, if \( \text{pre}(o) \subseteq s \) and \( \text{prv}(o) \subseteq s \). A plan is a valid plan from an initial state, \( s \), to a goal state, \( t \), if the first operator in the plan can be successfully executed in the initial state, the second operator can be successfully executed in the state resulting after the first operator etc. and also the goal state is subsumed by the state resulting after the last operator in the sequence. We use
the notion Valid(ω,s,t) to denote that the plan ω is a valid plan from the state,s, to the state, t.

**Definition 3.** The ternary relation Valid ⊆ Sequs(ℓ) × ℰ+ × ℰ+ is defined recursively, such that, for arbitrary operator sequence 〈o1,...,on〉 ∈ Sequs(ℓ) and arbitrary states s,t ∈ ℰ+, Valid(〈o1,...,on〉,s,t) iff either

1. n = 0 and t ⊆ s, or
2. n > 0, pre(oi) ⊆ s, pv(oi) ⊆ s and Valid(〈o2,...,on〉,s,t) iff either

A plan 〈o1,...,on〉 ∈ Sequs(ℓ) is a solution to a planning instance II = 〈V,ℓ,s0,sf〉 iff Valid(〈o1,...,on〉,s0,sf). The function, Result, returns the state resulting from executing a plan, i.e. t = Result(〈o1,...,on〉,s) if either t ∈ ℰ and Valid(〈o1,...,on〉,s,t) or t = 〈nil,...nil〉.

### 3. The IAO restriction

Since planning is PSPACE-complete also in the SAS+ formalism [5], in the general case, it is necessary to introduce further restrictions to be able to plan efficiently. The class of restricted problems considered in this article is a generalisation of the previously presented SAS+–IAO class [14], making it necessary to first recapitulate the restriction of this class. Most previous tractable classes of planning problems were defined using purely syntactical restrictions on the set of operators [3–5]. In contrast to this, the SAS+–IAO class is based on structural restrictions on the state-transition graph implicit in a planning problem. Since this graph may be of exponential size, it is too expensive to work on this graph directly, so an intermediate approach is taken. Restrictions are put on the domain-transition graph for each variable considered in isolation, that is, the graphs defined by projecting the full state-transition graph onto each variable domain separately. Although certain types of interference between these domain-transition graphs are taken into account, it is not possible to define all possible restrictions on the full state-transition graph in this way. That, however, is the trade-off taken to be able to plan and test restrictions in polynomial time.

All the definitions below implicitly assume some SAS+ structure Φ = 〈V,ℓ〉. We call an operator unary if it changes the value of exactly one state variable.

**Definition 4.** An operator, o ∈ ℰ, is unary iff there is exactly one variable, v ∈ V, such that pv(o)[v] ≠ u.

A defined value, 〈x, y, φ〉, for some variable, v, is a requestable value for v, if it may be necessary for v to take on the value x, in order to be able to change the value of another variable. The value x can be requestable for v in either of two cases: in the first case, the value x appears for v in the prevail-condition of an operator, o, changing the value of some other variable v′, which means that v must first get the value x before o can be executed to change the value of v′. In the second case, some operator, o, changing the value of another variable v′ also changes the value of v to or from the value x, which means that v must get the value x, either before changing the value of v′ or as a side effect of this. Formally, the set of requestable values of a variable is defined as follows:

**Definition 5.** For each variable, v ∈ V, the set ℰv of requestable values for the operator set, ℰ, is:

$$\mathcal{R}_v = \{pv(o)[v] | o \in \mathcal{E} \} \cup \{pr(o)[v], post(o)[v] | o \in \mathcal{E} \text{ and } o \text{ non-} \text{unary} \} \setminus \{u\}.$$  

It is an immediate observation that ℰv ⊆ ℰv for all v ∈ V.

For each state variable, we further define the domain-transition graph previously mentioned, that is, the graph of possible value transitions for this variable when considered in isolation from the other variables. The arcs in this graph are labelled by the operators corresponding to these transitions. Based on the domain-transition graph, we further define the reachability graph for an arbitrary subset, X, of the variable domain, that is, a graph with nodes restricted to the set, X, and where each arc denotes the existence of a path in the domain-transition graph between two nodes in X.

**Definition 6.** For each variable, v ∈ V, we define the corresponding domain-transition graph, Gv, as a directed labelled graph Gv = 〈Gv, ℋv〉, with vertex set ℰv and arc set, ℋv, such that, for all values x,y ∈ ℰv and operators o ∈ ℰ, 〈x,o,y〉 ∈ ℋv iff pr(o)[v] = x and po(o)[v] = y ≠ u. Further, for each subdomain, X ⊆ ℰv, we define the reachability graph for X as a directed graph GvX = 〈X, ℋvX〉, with vertex set, X, and arc set, ℋvX, such that for all x,y ∈ X, 〈x,y〉 ∈ ℋvX iff there is a path from x to y in Gv-

Alternatively, GvX can be viewed as the restriction to X ⊆ ℰv of the transitive closure of Gv but with unlabelled arcs. When speaking about a path in a domain-transition graph below, we will typically mean the sequence of labels, i.e. operators, along this path. We say that a path in Gv is via a set X ⊆ ℰv if each member of X is visited along the path, possibly as the initial or final vertex.

An operator is said to be irreplaceable if there is no other operator or sequence of operators which can replace it, which is made more precise in the following definition:

**Definition 7.** An operator, o ∈ ℰ, is irreplaceable wrt. a variable, v ∈ V, iff removing an arc labelled with o from Gv splits some component of Gv into two components.

We now have all the formal machinery required to define the three restrictions, I, A and O, which together define the SAS+–IAO class.
Definition 8. A SAS+ structure \( \langle \mathcal{V}, \mathcal{O} \rangle \) is:

1. Interference-safe iff every operator \( o \in \mathcal{O} \) is either unary or irreplaceable wrt. every variable, \( v \in \mathcal{V} \), it affects.
2. Acyclic iff \( G_v^o \) is acyclic for each variable, \( v \in \mathcal{V} \).
3. prevail-Order-preserving iff for each variable, \( v \in \mathcal{V} \), whenever there are two values \( x, y \in \mathcal{V}^v \), such that the graph \( G_v \) has a shortest path \( (o_1, \ldots, o_m) \) from \( x \) to \( y \) via some set \( X \subseteq \mathcal{V}^v \) of requestable values and it has any path \( \omega = (o_h^1, \ldots, o_h^k) \) from \( x \) to \( y \) via some set \( Y \subseteq \mathcal{V}^v \) of requestable values, such that \( X \subseteq Y \), there exists some subsequence \( \ldots, o_h^k, \ldots, o_h^1, \ldots \) of \( \omega \), such that \( \text{pr} \mathcal{V}(o_h^1), \ldots, \text{pr} \mathcal{V}(o_h^k) \) for \( 1 \leq k \leq m \).

The SAS+–IAO class is exactly the set of SAS+ problem instances satisfying all the restrictions I, A and O.

The motivation for interference-safeness is that if an operator changing the value of a variable, \( v \), in a desired way also has the side effect of changing the value of some other variable, \( v' \), then we know that there is no way to make the desired change to \( v \) without also changing \( v' \). This can be exploited in a planning algorithm to avoid search. That the reachability graph for the requestable values is acyclic is useful in the following way. If a variable, \( v \), must have a certain value in order to make a change in some other variable possible, then this value is a requestable value for \( v \). If there are several such necessary changes in other variables requiring \( v \) to pass through a number of its requestable values, then there is at most one possible order in which to pass through these values. This can also be exploited to avoid search. Finally, if an instance is prevail-order-preserving, then it can never be better to choose the longer of two alternative paths in a domain-transition graph, since the operators along the longer path always require at least the same requestable values of other variables and in the same order as the shorter one does.

Whether a SAS+ instance is an SAS+–IAO instance can be tested in polynomial time [21]. Moreover, a polynomial-time plan generation algorithm for the SAS+–IAO class has previously been presented [14, 21] and this algorithm will be referred to later in this article under the name PlanIAO.

4. The planning algorithm

In this section we present a formally correct extension of the SAS+–IAO algorithm [14], which is capable of generating plans for the LEGO car factory. The basic idea is to partition the original SAS+ instance into two separate instances, both being SAS+–IAO instances. The first of these instances can then be solved in polynomial time and its solution constitutes a skeleton to be filled in by solving subproblems from the second instance. This process is referred to as interweaving and can be viewed as a restricted variant of the more general concept, refinement, as used in hierarchical state abstraction [22, 23]. In fact, the whole method we use can be viewed as a restricted variant of two-level state abstraction. However, while the state abstraction method in general is not yet well understood — it can sometimes speed up planning exponentially [23] and sometimes slow down planning exponentially [24] — our, more restricted, method is provably correct and runs in polynomial-time.

First we define the restriction of an SAS+ structure to a subset of its variables:

Definition 9. Let \( \langle \mathcal{V}, \mathcal{O} \rangle \) be a SAS+ structure. Then, the restriction to a variable set \( \mathcal{V}' = \{v_{i_1}, \ldots, v_{i_k}\} \subseteq \mathcal{V} \) of a state \( s \in \mathcal{V}^\mathcal{O} \), an operator, \( o \in \mathcal{O} \) or an operator set, \( \mathcal{O}' \subseteq \mathcal{O} \), is denoted \( s \cap \mathcal{V}' \), \( o \cap \mathcal{V}' \) and \( \mathcal{O} \cap \mathcal{V}' \) respectively, and is defined as:

- \( s \cap \mathcal{V}' = \{s[i_{i_1}], \ldots, s[i_{i_k}]\} \)
- \( o \cap \mathcal{V}' = (\text{pre} o) \cap \mathcal{V}', \text{post} o \cap \mathcal{V}', \text{pr} V o \cap \mathcal{V}' \)
- \( \mathcal{O} \cap \mathcal{V}' = \{o \cap \mathcal{V}' | o \in \mathcal{O}\} \)

This concept is extended to SAS+ structures and SAS+ instances by component-wise application, that is (\( \mathcal{V}', \mathcal{O} \cap \mathcal{V}' \))\( \mathcal{V}' = (\mathcal{V}', \mathcal{O} \cap \mathcal{V}') \) and (\( \mathcal{V}, \mathcal{O}, s, t \))\( \mathcal{V}' = (\mathcal{V}', \mathcal{O} \cap \mathcal{V}', s \cap \mathcal{V}', t \cap \mathcal{V}') \).

The concept of restriction can be used to split a SAS+ instance into smaller subinstances, by partitioning the set of variables and taking the restrictions to each part of the original instance. In particular, we are interested in partitioning the variable set \( \mathcal{V} \) of a SAS+ structure, \( \Phi = (\mathcal{V}, \mathcal{O}) \), into two sets, \( \mathcal{V}_1 \) and \( \mathcal{V}_2 \), such that \( \mathcal{V}_1 \) is independent of \( \mathcal{V}_2 \). This means that a plan for the restriction of \( \Phi \) to \( \mathcal{V}_1 \) does not affect any variables in \( \mathcal{V}_2 \) and can be executed regardless of the values of these variables, while a plan for the restriction of \( \Phi \) to \( \mathcal{V}_2 \), likewise, does not affect any variables in \( \mathcal{V}_1 \) but may require such variables to take on requestable values in order to be executable. That is, there is a one-or-one dependence between the two partitions.

Definition 10. Let \( \Phi = (\mathcal{V}', \mathcal{O}) \) be a SAS+ structure and let \( \mathcal{V}_1, \mathcal{V}_2 \) be disjoint subsets of \( \mathcal{V} \). Then, \( \mathcal{V}_1 \) is independent of \( \mathcal{V}_2 \) iff:

- every operator, \( o \in \mathcal{O} \), affecting some variable in \( \mathcal{V}_1 \) satisfies \( \text{pre} o \cap \mathcal{V}_2 = \text{post} o \cap \mathcal{V}_2 = \text{pr} V o \cap \mathcal{V}_2 = (\ldots, o, \ldots) \);
- every operator, \( o \in \mathcal{O} \), affecting some variable in \( \mathcal{V}_2 \) satisfies \( \text{pre} o \cap \mathcal{V}_1 = \text{post} o \cap \mathcal{V}_1 = (u, \ldots, u) \).

We further adopt the term reachability from automatic control, with an analogous meaning. A SAS+ structure is reachable if there exist plans for going from any state to any other state in the state space of this instance.

Definition 11. Let \( \Phi = (\mathcal{V}', \mathcal{O}) \) be a SAS+ structure. Then \( \Phi \) is reachable iff for any two states, \( s, t \in \mathcal{V}' \), the planning problem instance \( (\mathcal{V}', \mathcal{O}, s, t) \) is solvable.
A planning problem instance \( \langle \mathcal{F}, \mathcal{C}, s_0, s_r \rangle \) is reachable iff the corresponding structure, \( \langle \mathcal{F}, \mathcal{C} \rangle \) is reachable. It is an immediate consequence that a reachable instance has a strongly connected state-transition graph.

We can now show that if a SAS\(^+\) instance, \( \Pi \), is split into two independent instances with certain properties, then it is sufficient to solve these subinstances in order to solve the original instance.

**Theorem 1.** Let \( \Pi = \langle \mathcal{F}, \mathcal{C}, s_0, s_r \rangle \) be a SAS\(^+\) instance and let \( \mathcal{F} = \langle \mathcal{F}_1, \mathcal{F}_2 \rangle \) be a partitioning of \( \mathcal{F} \). If \( \mathcal{F}_1 \) is independent of \( \mathcal{F}_2 \), \( \Pi \cap \mathcal{F}_1 \) is reachable and \( \Pi \cap \mathcal{F}_2 \) is solvable, then \( \Pi \) is solvable.

**Proof.** Suppose there exists a plan, \( \omega = \langle o_1, \ldots, o_n \rangle \), solving \( \Pi \cap \mathcal{F}_2 \). Since \( \Pi \cap \mathcal{F}_1 \) is reachable, there must be some plan, \( \omega_0 \), solving the SAS\(^+\) instance \( \langle \mathcal{F}_1, \mathcal{C}, s_0, \text{Result}(\omega_0) \rangle \) \( \cap \mathcal{F}_1 \). Obviously, \( o_1 \) is executable in the state \( \text{Result}(\omega_0, s_0) \).

Moreover, recursively define the plans \( \omega_k \), where \( 1 \leq k \leq n - 1 \), such that \( o_k \) is a solution to the instance:

\[
\langle \mathcal{F}_1, \mathcal{C}, \text{Result}(\langle o_0, \ldots, o_{k-1} \rangle, s_0), \text{Result}(o_k) \rangle \cap \mathcal{F}_1
\]

Finally, let \( \omega_n \) be a solution to the instance:

\[
\langle \mathcal{F}_1, \mathcal{C}, \text{Result}(\langle o_0, \ldots, o_{n-1} \rangle, s_0), s_r \rangle \cap \mathcal{F}_1
\]

Since \( \Pi \cap \mathcal{F}_1 \) is reachable, then the plans \( \omega_0 \ldots \omega_n \) must exist. Now, let us consider the plan, \( \omega' = \langle o_0, o_0, o_1, \ldots, o_{n-2}, o_{n-1} \rangle \). By the construction of \( \omega_0 \ldots \omega_n \), all pre- and prevail-conditions of \( o_1 \ldots o_n \) are satisfied because \( \omega = \langle o_1, \ldots, o_n \rangle \) is a valid plan solving \( \Pi \cap \mathcal{F}_2 \) by assumption. Moreover, all pre- and prevail-conditions of the operators in \( \langle o_0, \ldots, o_n \rangle \) are satisfied because \( \langle o_0, \ldots, o_n \rangle \) is a valid plan and \( \mathcal{F}_1 \) is independent of \( \mathcal{F}_2 \). It remains to show that \( s_r \subseteq \text{Result}(\omega', s_0) \).

This follows immediately, since we know that \( \langle o_1, \ldots, o_n \rangle \) is a valid plan for \( \Pi \cap \mathcal{F}_2 \) and \( \langle o_0, \ldots, o_n \rangle \) is a valid plan for \( \Pi \cap \mathcal{F}_1 \) and \( \mathcal{F}_1 \) is independent of \( \mathcal{F}_2 \).

Fig. 1 shows a planning algorithm based on the proof of Theorem 1. The sets \( \mathcal{F}_1 \) and \( \mathcal{F}_2 \) are such that both \( \Pi \cap \mathcal{F}_1 \) and \( \Pi \cap \mathcal{F}_2 \) satisfy the IAO restriction in the previous section, and \( \mathcal{F}_1 \) is independent of \( \mathcal{F}_2 \). The procedure \texttt{PlanIAO} used in this algorithm is the planning algorithm previously presented for the SAS\(^-\)IAO planning problem [14]. It is obvious from the proof of Theorem 1 that the algorithm is correct, and since \texttt{PlanIAO} is polynomial, the resulting algorithm is polynomial.

The first step in the algorithm is to find a plan \( \langle o_1, \ldots, o_n \rangle \), from \( s_0 \) to \( s_r \), when only state variables from the set \( \mathcal{F}_2 \) and only operators affecting such variables are taken into account (line 2 in Fig. 1). This results in an incomplete plan that may not be executable due to unsatisfied prevail-conditions. The second step, referred to as interleaving, attempts to fill in these 'gaps' with subplans satisfying the remaining prevail-conditions (lines 3–7 in Fig. 1). The interleaving process tests for each operator in the incomplete plan, whether its prevail-conditions are satisfied at this point in the plan, i.e., whether the prevail-condition for the operator, \( o_{k+1} \), is satisfied in the state, \( s_k \), which is the state reached when executing the operators in the plan \( \langle o_0, o_1, o_2, \ldots, o_{k-2}, o_{k-1}, o_{k+1} \rangle \). If the prevail-condition is not satisfied, then a plan, \( \omega_k \), achieving the desired prevail-condition is developed, using only operators in the set \( \mathcal{C} \cap \mathcal{F}_1 \). That such a plan exists is obvious since \( \Pi \cap \mathcal{F}_1 \) is reachable. In the last step (line 8 in Fig. 1), the plans constructed during the interleaving process are spliced into the original incomplete plan, resulting in a complete plan that solves the original problem and is executable.

**5. The planning tool**

Fig. 2 shows the basic modules of the planning tool and the information flow between them. The two main modules are the planner, which generates operation plans, and the translator, which translates the plans into GRAFCET charts. GRAFCET is an IEC-standard [17] graphical language for process control based on Petri nets [25]. The GRAFCET chart produced is then compiled into code for a programmable logic controller (PLC) using a commercial compiler [26] (not shown in the figure) and loaded into the PLC which controls the process. Both the planner and the translator are based on a model of the process to be controlled, but the model provides different views of the process for the two modules. The planner view of the model describes the process in the SAS\(^+\) language, that is, in terms of states and state-changing operators modelled with pre-, post- and prevail-conditions, while the translator view provides information on how to translate each operator in the planner view into a corresponding GRAFCET fragment. The first of these views is, thus, a more abstract model which ties together the functioning...
of the various parts of the process, while the second view provides fragmented knowledge of how to translate the abstract operator descriptions into actual sensor readings and actuator signals.

In principle, one could plug in any planning module as long as it could be supported by a model which also provides a corresponding view for the translator. In particular, any planner designed for the full SAS$^+$ language, or some sublanguage thereof, would work. Since our main concern is efficient planning, we focus on using efficient planners for sublanguages, which means that the choice of planner must depend on the characteristics of the process at hand. In the LEGO-car-factory example, to be presented shortly, we use the planner described in Section 4 and the details on the implementation of the planner are described by Kvarnström [27]. The translator is straightforward and will not be described in detail here, but a graphical example is provided in Fig. 3 (Russian [28] gives a more detailed account of the translator module).
6. The LEGO car factory

Our application example is an automated assembly line for LEGO cars, [16] which is used for undergraduate laboratory sessions in digital control at the Department of Electrical Engineering at Linköping University. The students are faced with the task of writing a program to control this assembly line using GRAFCET. The LEGO car factory is a realistic miniature version of a real industrial process in many respects. Finding a polynomial-time planning algorithm capable of planning for the LEGO-car factory has, thus, been a long-standing goal in our research.

The main assembly operations for building a LEGO car are shown in Fig. 4. However, this is an abstracted view from the point of the workpiece, i.e. the car, and the actual operation plan to achieve the effect of these two abstract assembly operations is much more complex. The assembly line consists of two similar halves, the first mounting the chassis parts on the chassis and the second mounting the top. The first half of the LEGO car factory is presented in Fig. 5. The chassis' are initially stored upside down in a stack in the chassis magazine (cm), and the lowermost chassis can be pushed out onto the conveyor belt by the chassis feeder (c-feeder). The chassis is then transported to the chassis-parts magazine (cpm) where, analogously to the chassis magazine, the chassis parts are stored in a stack such that the lowermost set of chassis parts can be pushed out onto the current chassis by the chassis-parts feeder (cp-feeder). The conveyor belt runs continuously, so to stop a chassis one does not stop the belt, but pushes out a stopper bar in front of the chassis to hold it fixed at the desired position, the belt thus sliding under the chassis. At the chassis-parts magazine, for instance, the stopper bar, cpm-stop, must be pushed out across the conveyor belt before the chassis reaches the chassis-parts magazine. When the chassis parts have been pushed out onto the chassis, the feeder is first retracted and then the stopper bar is retracted, allowing the chassis to move on. The chassis is then transported to the chassis press (cp), where it is once again stopped by a stopper bar (cp-stop) and the chassis parts, put onto the chassis at the previous workstation, are pressed tight onto the chassis by the chassis press. The chassis is then once again released and moves on to the end of the first conveyor belt, where it is turned into an upright position and moved into the second half of the factory, which is shown in Fig. 6.

Here the chassis enters a chassis lift (cl) which moves the
chassis onto the second conveyor belt. The second half of the factory is analogous to the first half, consisting of two workstations, the first one placing a top on the car from the top magazine \((tm)\) and the second one pressing the top tight onto the car. Finally, the finished car is pushed off the belt, sideways, into the buffer storage \((st)\).

Fig. 7 shows one of the work-stations in more detail, namely the one where the top is put onto the chassis \((tm\) in Fig. 6). The chassis is held fixed at the top storage \((A)\) by the stopper bar \((B)\). The tops are stored in a stack and the feeder \((C)\) is used to push out the lowermost top onto the chassis. When the top is on the chassis, the feeder is withdrawn and then the stopper bar is withdrawn, thus allowing the chassis to move on to the next work-station.

### 7. The planner-view of the factory model

To build the planner view of the planning-tool model for this process, we first introduce the state variables shown in Table 1, together with their corresponding value domains.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>pos</td>
<td>(cm, cpm, cp, ts, cl, ocvB, tm, tp, sf, st)</td>
</tr>
<tr>
<td>turner</td>
<td>(A, B)</td>
</tr>
<tr>
<td>cp-status</td>
<td>off, on, pressed</td>
</tr>
<tr>
<td>t-status</td>
<td>off, on, pressed</td>
</tr>
<tr>
<td>c-status</td>
<td>prepared, not-prepared</td>
</tr>
<tr>
<td>cp-press</td>
<td>down, up</td>
</tr>
<tr>
<td>t-press</td>
<td>down, up</td>
</tr>
<tr>
<td>cliff</td>
<td>down, up</td>
</tr>
<tr>
<td>c-feeder, cp-feeder,</td>
<td>(ext, rtr)</td>
</tr>
<tr>
<td>t-feeder, st-feeder</td>
<td>(ext, rtr)</td>
</tr>
<tr>
<td>cpm-stop, cp-stop,</td>
<td>(ext, rtr)</td>
</tr>
<tr>
<td>tp-stop</td>
<td>(ext, rtr)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operator</th>
<th>Pre</th>
<th>Post</th>
<th>Prevail</th>
</tr>
</thead>
<tbody>
<tr>
<td>cm2cpm</td>
<td>pos = cm</td>
<td>pos = cpm</td>
<td>(c-feeder = ext, cp-feeder = rtr, cpm-stop = ext)</td>
</tr>
<tr>
<td>cpm2cp</td>
<td>pos = cpm</td>
<td>pos = cp</td>
<td>(cpm-stop = rtr, cp-stop = ext, cp-press = up)</td>
</tr>
<tr>
<td>c2cpm</td>
<td>pos = cp</td>
<td>pos = cp</td>
<td>(turner = A, cp-stop = rtr, cp-press = up)</td>
</tr>
<tr>
<td>t2ts</td>
<td>pos = ts</td>
<td>pos = cp</td>
<td>(turner = B, cliff = down)</td>
</tr>
<tr>
<td>cl2ocvB</td>
<td>pos = cl</td>
<td>pos = ocvB</td>
<td>(clift = up)</td>
</tr>
<tr>
<td>ocvB2tm</td>
<td>pos = ocvB</td>
<td>pos = tm</td>
<td>(tm-stop = ext, t-feeder = rtr)</td>
</tr>
<tr>
<td>tm2tp</td>
<td>pos = tm</td>
<td>pos = tp</td>
<td>(tm-stop = rtr, tp-stop = ext, t-press = up)</td>
</tr>
<tr>
<td>tp2sf</td>
<td>pos = tp</td>
<td>pos = sf</td>
<td>(tp-stop = rtr, st-feeder = rtr, t-press = up)</td>
</tr>
<tr>
<td>sf2St</td>
<td>pos = sf</td>
<td>pos = st</td>
<td>(st-feeder = ext)</td>
</tr>
<tr>
<td>prepare-chassis</td>
<td>c-status = not-prepared</td>
<td>c-status = prepared</td>
<td>(c-feeder = rtr)</td>
</tr>
<tr>
<td>put-cp</td>
<td>cp-status = (off)</td>
<td>cp-status = on</td>
<td>(pos = cp, cp-feeder = ext)</td>
</tr>
<tr>
<td>press-cp</td>
<td>cp-status = on</td>
<td>cp-status = pressed</td>
<td>(cp-press = down, pos = cp)</td>
</tr>
<tr>
<td>put-top</td>
<td>t-status = (off)</td>
<td>t-status = on</td>
<td>(pos = tm, t-feeder = ext)</td>
</tr>
<tr>
<td>press-top</td>
<td>t-status = on</td>
<td>t-status = pressed</td>
<td>(pos = tp, t-press = down)</td>
</tr>
</tbody>
</table>
Most of these variables correspond to stopper and feeder bars etc. and have the obvious names and values (\textit{ext} for extended and \textit{rtr} for retracted). The variable \textit{pos} gives the position of the chassis, with discrete values as indicated in Figs. 5 and 6, and the variable \textit{turner} tells if the turner (\textit{ts} in Fig. 5) is turned towards the first half of the factory (A) or towards the second half of the factory (B). Moreover, the two variables \textit{cp-status} and \textit{t-status} give the status of the chassis parts and the top, respectively, while the variable \textit{c-status} denotes the status of the chassis and is mainly needed since we have no sensor detecting if the chassis is just outside the chassis magazine.

Using these variables, we define SAS$^+$ operators corresponding to the possible actions that can be executed in the factory. Table 2 shows the workpiece-related actions, that is, the actions changing the position of the chassis or changing its status, and Table 3 shows the operators for controlling the turner, the lift and the presses. Finally, there must be operators for extending and retracting the stopper and feeder bars. These are not shown explicitly here, but are named systematically. For instance, the operators corresponding to the chassis feeder are denoted \textit{extend-c-feeder} and \textit{retract-c-feeder}. The pre-condition is that \textit{c-feeder} $=$ \textit{rtr}, the post-condition is that \textit{c-feeder} $=$ \textit{ext} and there is no prevail-condition.

It should be noted that while some operators correspond directly to real actions in the factory, like the operator for extending a particular feeder bar, other operators correspond to actions which should rather be viewed as \textit{events}. For instance, since the conveyor belt runs continuously it is not possible to directly execute the operator, \textit{cm2cpm}, which moves the chassis from the chassis magazine to the chassis-parts magazine. In the real world, this is an event which is automatically initiated as soon as the chassis is pushed out onto the conveyor belt and terminates successfully only if the stopper bar at the chassis-parts magazine is extended when the chassis reaches it. For this particular process, it is sufficient to view such events as actions and model them as operators, but some care must be taken in the modelling. For instance, the prevail-condition of this particular operator is that the stopper bar is extended, that is, we must extend the stopper bar already before we push out the chassis onto the conveyor belt. Although this means extending the stopper bar somewhat earlier than is strictly necessary, it is not likely to degrade performance in practice; even if we want to pipeline the production of cars, we would, for safety reasons, probably not schedule two chassis so that they appear on the same belt at the same time without at least one extended stopper bar between them.

8. Planning for the LEGO car factory

The model of the LEGO-car factory presented in the previous section is an SAS$^+$ structure, $\Phi = (\mathcal{V}, \mathcal{E})$. In order to choose a suitable planning algorithm for this structure, we have to examine the characteristics of it. It can be verified that the model does not satisfy the restrictions of the SAS$^+$–IAO class [14] or any of the other previously presented tractable SAS$^+$ subclasses [5]. However, it is easily seen that only the operators in Table 2 have defined prevail-conditions. We can thus divide the state variables into two sets: $\mathcal{V}_2$ containing those variables affected by operators with defined prevail-conditions, i.e., the operators in Table 2, and $\mathcal{V}_1$ containing those variables affected by other operators. Obviously, these two sets form a partitioning of the set of state variables having the following values:

$\mathcal{V}_1 = \{\text{turner}, \text{cp-press}, \text{t-press}, \text{clift}, \text{c-feeder}, \text{cp-feeder}, \text{t-feeder}, \text{st-feeder}, \text{cpm-stop}, \text{cp-stop}, \text{tm-stop}, \text{tp-stop}\}$

$\mathcal{V}_2 = \{\text{pos}, \text{cp-status}, \text{t-status}, \text{c-status}\}$

These variables thus define two restricted substructures, $\Phi \cap \mathcal{V}_1$ and $\Phi \cap \mathcal{V}_2$ of $\Phi$ and it can be further verified that the first of these structures is reachable and that $\mathcal{V}_1$ is
independent of $\forall \exists$. Moreover, both these substructures are SAS$^+$–IAO structures, so the planning algorithm presented in Fig. 1 can be used to plan for the LEGO-car factory.

Depending on how we choose the initial state and the goal state, we can plan for different cases. As an example we show a plan for normal operation, where the goal is a fully assembled LEGO car and the initial state is a factory in ‘resting’ state, that is, the chassis is placed in the chassis magazine ($pos = cm$, $c-status = not-prepared$), there are no chassis parts on the chassis ($cp-status = off$) and there is no top on the chassis ($t-status = off$). Additionally, the turner is turned towards the first half of the factory ($turner = A$), all feeders and stopper bars are retracted and the chassis press, the top press and the chassis lift are in their down position. The goal state is that the chassis should be in the buffer storage ($pos = st$) and the top and chassis parts should be pressed onto the chassis ($cp-status = pressed$ and $t-status = pressed$). All other state variables are left undefined in the goal states, meaning that they are allowed to have any value at the end of the plan, that is, we do not care about their final values.

Applying the algorithm in Fig. 1 results in the plan shown as a directed graph in Fig. 8. The subgraph consisting of those operators that are connected by solid arrows, is the plan skeleton resulting from the first step in the algorithm (line 2 in Fig. 1). This plan cannot be executed due to unsatisfied prevail-conditions. For example, the operator, $cm2cpm$, cannot be executed in its current position in the plan, since the chassis feeder is retracted in the initial state and not affected by the previous operator $prepare-chassis$, while it has to be extended in order to execute the operator, $cm2cpm$. In order to solve such problems, the interweaving process (lines 3–7 in Fig. 1) adds those operators connected by dashed arrows in Fig. 8. The final resulting plan thus consists of all operators in this figure, and they must be executed in the order indicated by the dashed arrows.

In this case, we use a partial goal state, where some variables are allowed to have any final value. We could also choose to specify the desired final value for all variables, in order to have a well-defined final state. Allowing partial goal states is useful in some cases, however. For instance, if we are not certain what states are possible and only care about the value of some variables, or if we do not know what initial state will be preferred for the subsequent plan. Similarly, it is possible to have a partial initial state, that is, leaving some variables undefined in the initial state, signalling that the values of these are unknown. In order to plan in such cases, however, we need operators that can change a variable from the undefined value, that is, from any value, to some defined value. No such operators are defined for this particular application, but they are allowed in the SAS$^+$ formalism.

In addition, the example above showed a plan for the normal operation of the plant. However, the planner could also be invoked to plan from other initial states, for instance, the state we happened to end up in after a breakdown or emergency stop. Similarly, we could have alternative goals, like producing cars with or without a top, for instance.

The assembly process described above is admittedly simple. However, one could easily imagine more complex variants of it, with considerably more flexibility and possibilities. A simple extension for handling errors could be to have separate buffer stores for correctly and incorrectly assembled cars, respectively, or some feedback device that moves the partially assembled car back to a suitable workstation for redoing the operation that failed.

9. Discussion

Tsatsoulis and Kashyap [29] call planning ‘one of the most underused techniques of AI’ in the context of manufacturing. They list a number of areas within industry where planning could be applied, but where no or very few attempts have been made at such applications. There are some examples of such work in the recent literature, however, including the following examples. Nau et al. [30] have studied the computational complexity of the domain-specific problem of machining parts, and also developed a planner for this problem. However, their planner is not based on standard AI planning techniques, but is a specialised algorithm tailored to the needs of the particular application. Kis and Vánca [31] analyze the computational complexity of manufacturing planning modelled with STRIPS operators, but their planning system is chiefly based on a combination of expert systems and genetic programming [32]. Munós-Avila and Weberskirch [33] present a system for planning the manufacturing of workpieces and they seem to use STRIPS-style operators. Their planner uses standard techniques from AI planning, but it is the technique of case-based planning and plan adaptation rather than the techniques for producing plans ‘from scratch’.

Our work differs from these examples in two aspects. First, we use a general-purpose AI planner, designed for restricted languages, but not specifically tailored to manufacturing and assembly problems. Second, although applicable for generating assembly plans, our system is primarily intended for error-recovery planning. It is also important to note that, although the interweaving planner used in this

<table>
<thead>
<tr>
<th>Operator without prevail-conditions</th>
<th>Pre</th>
<th>Post</th>
<th>Prevail</th>
</tr>
</thead>
<tbody>
<tr>
<td>A2B</td>
<td>$turner = A$</td>
<td>$turner = B$</td>
<td>–</td>
</tr>
<tr>
<td>B2A</td>
<td>$turner = B$</td>
<td>$turner = A$</td>
<td>–</td>
</tr>
<tr>
<td>cl-down</td>
<td>$clift = up$</td>
<td>$clift = down$</td>
<td>–</td>
</tr>
<tr>
<td>cl-up</td>
<td>$clift = down$</td>
<td>$clift = up$</td>
<td>–</td>
</tr>
<tr>
<td>cp-press-down</td>
<td>$cp-press = up$</td>
<td>$cp-press = down$</td>
<td>–</td>
</tr>
<tr>
<td>cp-press-up</td>
<td>$cp-press = down$</td>
<td>$cp-press = up$</td>
<td>–</td>
</tr>
<tr>
<td>t-press-down</td>
<td>$t-press = up$</td>
<td>$t-press = down$</td>
<td>–</td>
</tr>
<tr>
<td>t-press-up</td>
<td>$t-press = down$</td>
<td>$t-press = up$</td>
<td>–</td>
</tr>
</tbody>
</table>
application was developed with the purpose of planning efficiently for the LEGO car factory, we have also developed other polynomial-time planners capable of planning for this application, e.g. the 3S planner [34]. This planner was not designed with any particular application in mind, but stems from purely theoretical considerations.

The reason that we could not apply the SAS$^+$–IAO planner directly is that it does not allow cycles involving more than one requestable value. This limitation is to some extent overcome in the new planner, since the interweaving process allows certain restricted types of such cycles. However, although this is sufficient for modelling the LEGO car factory and probably also for modelling a number of other interesting processes, it should not be expected to suffice for the large share of real applications. This means that further extensions and/or modifications will be required, and it is, thus, a topic of future research to investigate how to handle other, more complex forms of cycles. Further, it should not be forgotten that the planning tool is not dependent on planners being based on the SAS$^+$–IAO class, so other planners could also be used whenever required.

Although the planner for the SAS$^+$–IAO problem produces partial-order plans, having some potential for parallelizing operators, the subsequent interweaving process generates a total-order plan as output. Hence, the current planning tool can only produce plans where all operations must be executed in sequence. This is not an inherent limitation in the factory, however, so our plans for future research include modifying the planning algorithm and planning tool to produce plans which allow for parallel execution of operators and thus increase the throughput by pipelining. Although the problem of producing optimal parallel plans is known to be very difficult in the general case [35], we believe this particular application to have sufficient structure to allow for efficient parallelization of plans. The future plans also include closing the feedback loop by continuously monitoring the assembly line to detect deviations from the expected state when executing plans. In such cases, plan execution should be stopped and the current actual state of the process be fed back to the planner, which can then produce a plan for recovering from this state.

10. Conclusions

We have applied our previous results on polynomial-time planning to an application example in automatic control — an assembly line for LEGO cars. Since none of the restricted planning languages used by our previous polynomial-time planners was sufficiently expressive for modelling this application, we had to modify one of the planners. This was done by using one of the previous planners, the planner for the SAS$^+$–IAO class, as a subprocedure in a provably correct and efficient two-level hierarchical planner. Although developed for this application, the planner is a general-purpose planner, not specifically tailored to the particular application. This planner is used in a planning tool which works in the following way. The planning algorithm produces a plan, based on a model of the factory, and then a translation module converts this plan into an equivalent GRAFCET chart. Finally, a GRAFCET compiler is used to compile this chart into code for the programmable logic controller that controls the factory.

This paper is an example where an attempt to apply results, that were theoretical in origin, to an application provided feedback for modifying the theory in a way that allowed for application solving, as well as extending the theoretical results in a non-application-dependent way.

Acknowledgements

Jonas Kvarnström implemented the planning algorithm and Fabio Russian implemented the program for translating plans into GRAFCET charts. The research presented in this article was sponsored by the Swedish Research Council for Engineering Sciences (TFR) under grants dnr. 92-143, 93-291 and 95-731.

References


I. Klein et al. / Artificial Intelligence in Engineering 13 (1999) 69–81


