Sequence Planning Using Multiple and Coordinated Sequences of Operations

Kristofer Bengtsson, Patrik Bergagård, Carl Thorstensson, Bengt Lennartson, Knut Åkesson, Chengyin Yuan, Sajed Miremadi, and Petter Falkman

Abstract—The sequential behavior of a manufacturing system results from several constraints introduced during the product, manufacturing, and control logic development. This paper proposes methods and algorithms for automatically representing and visualizing this behavior from various perspectives throughout the development process. A new sequence planning approach is introduced that uses self-contained operations to model the activities and execution constraints. These operations can be represented and visualized from multiple perspectives using a graphical and formal language called Sequences of Operations (SOPs).

The operations in a manufacturing system are related to each other in various ways, due to execution constraints expressed by operation pre- and post-conditions. These operation relations include parallel, sequence, arbitrary order, alternative, and hierarchy relations. Based on the SOP language, these relations are identified and visualized in various SOPs and sequences. A software tool, Sequence Planner, has been developed, for organizing the operations into SOPs that visualize only relevant operations and relations.

Note to Practitioners—This paper proposes methods and algorithms for a new sequence planning approach in which sequences are automatically created based on the relations among operations instead of having to be manually constructed. Using various views, the sequences of operations related to, for example, part flow, robot operations, and operator tasks, can be visualized. The use of various views helps the user better understand the relations between cell control and mechanical design, and between product design and total system behavior.

Index Terms—Automation design, control logic design, formal methods, sequence planning, sequences of operations.

I. INTRODUCTION

The famous motto “There is more than one way to do it” [27] was influential when the software language Perl was being developed. Perl’s inventor, Larry Wall, a trained linguist, believed that sometimes another phrasing could make the meaning of a design clearer. The truth of this motto is even more obvious in the research field of information visualization, where Bertin stated: “A graphic is no longer drawn once and for all: it is constructed and reconstructed (manipulated) until all the relationships which lie within it have been perceived, . . . A graphic is never an end in itself: it is a moment in the process of decision making” [23].

When developing an automation system, many engineering disciplines work together to create a functional system. Such an effort involves product designers, process planners, tool designers, robot programmers, automation engineers, logistic planners, etc. Most of these engineers have differing views of the system and the design process. It is therefore important to be able to represent design information from multiple perspectives, to make it clear to everyone in the design team.

A common method for visualizing complex information is to use multiple views to represent different projections of the information [23]. This research area, which is called coordinated and multiple views (CMVs) [17], has identified many interesting techniques and guidelines [2]. The present paper presents a tool that can be used to visualize and manage information related to sequence planning in multiple views. In particular, two CMV research areas are emphasized in this tool: data processing and preparation, and view generation.

A. Sequence Planning

The purpose of sequence planning, when developing an automation system, is to determine in what order operations could or should execute. Sequence planning integrates high-level requirements, such as cycle time and product quality issues, with low-level sequencing constraints, such as mutual exclusion, safety concerns, and actuator actions. In the present research, the operations describe the logical behavior of tasks executed by an automation system to accomplish a specific objective. The relations among operations are defined by execution conditions governing when and how an operation can execute, for example, to ensure that one particular operation always precedes another.

The challenges involved in planning operation sequences are addressed in many research areas, for example, project management [11], business process automation [6], product assembly planning [28], manufacturing task planning [18],...
workflow scheduling [8], computer-aided manufacturing [14], computer-aided process planning [13], and control design [19]. Academics have focused mostly on optimization-based planning problems, but the industrial impact has been quite limited so far, probably due to the complexity of solving real problems. The industry focus has instead been on representing and visualizing sequences and tasks and on simulating them.

B. Gantt Chart

Since the operation specification of an automation system is central [3], all involved stakeholders must understand the operation specification and how various design decisions influence it. The most widespread industrial tool for specifying operations is probably the Gantt chart [29], which is easy to use and understand and intuitive to work with [9]. However, it was soon recognized that planning complex, large-scale production systems was too complicated for the Gantt chart [29]. The chart does not really represent logical relations among system elements, but rather their durations and start and stop times. For example, Gantt charts cannot handle alternatives and arbitrary order or represent interdependent and network-like relations [29].

Since sequence planning is fundamentally logical, planning using Gantt charts alone can increase the development problems. Today, it is fairly common for certain sequences to be specified early in the development process to avoid complexity and to fit them into a Gantt chart. Indirectly, this over specifies all operations. This is a problem, especially in early development phases and platform development [7], when it is important to retain as much freedom as possible in terms of parallelism, and to increase flexibility, adaptivity, and optimality. Moreover, early specification of sequences to avoid complexity can complicate the introduction of changes in the product or manufacturing process later on in the development process.

C. Method for Planning Operations and Sequences

These drawbacks tend to result in inconsistent and contradictory understandings of system behavior. Therefore, a new method for planning operation sequences has been proposed by Lennartson et al. [12] and Bengtsson et al. [4], who present a graphical and formal language, Sequences of Operations (SOPs).

The SOP language is based on self-contained operations that include only relevant conditions as to when and how every operation can execute, i.e., no information is stored dealing explicitly with a particular sequence, as in a Gantt chart. The relations within a set of operations can then be identified and visualized graphically from various perspectives, based on the constraints on each operation. Examples of these perspectives are operation sequences related to part flow through manufacturing, the execution sequence of a specific robot, and operations related to a specific safety system.

A multiplicity of perspectives helps the user better understand the relation between cell control and mechanical design, between product design and total system behavior. The suggested method considers not only the industrial need for easy visualization and improved understanding of sequential behavior, but also the need to optimize and verify the system (e.g., [10] and [26]).

The SOP language focuses on how to organize, prepare, and process data related to manufacturing operations, to be able to visualize the information in multiple views. Using the SOP language and earlier research findings, this paper focuses on how to generate these multiple views, covering both necessary CMV research areas.

This paper, together with an earlier one by the same authors [5], presents algorithms for organizing operations into sequences, identifying and visualizing the relations between them automatically, and coordinating the various SOPs. These algorithms are implemented in a tool called Sequence Planner (SP) [16], which is used not only to visualize the operations in various coordinated views to help engineers plan the sequences, but also to verify, synthesize, and optimize the sequences.

The next section introduces the formal language, Sequences of Operations (SOPs). Section III demonstrates how to identify relations among operations, while Section IV presents a method for automatically visualizing sequences. The SP tool is introduced in Section V, which also describes how to manage the coordination among operations and SOPs.

II. SEQUENCES OF OPERATIONS (SOPs)

This section presents the method introduced by Lennartson et al. [12] and Bengtsson et al. [4] together with an example that will be referred to throughout the paper.

The core object of the graphical language SOPs is the operation. An operation specifies, at various levels of detail, an action or task executed by an automation system. An operation can describe, for example, the assembly of a complete product or the action of sending a signal to an actuator. This is handled by including suboperations in an operation. The level of detail of an operation specification is related to the purpose of the SOP model. Sometimes only high-level operations are relevant, but in other cases, for example, when implementing the operations in a control system, greater detail is needed.

An operation is visualized in the SOP language as in Fig. 1. Each operation includes a set of conditions, describing when and how the operation will execute. The preconditions specify when an operation can start to execute and may include guard expressions related to other operations, the state of a resource, and product or safety interlocking expressions. The preconditions also cover actions describing what should happen when the operation starts.

The operation will continue to execute until the postconditions are satisfied. Like the preconditions, the postconditions also include both guards and actions. The algorithms presented here use a formal mathematical model of the operations based on automata.

A. A Formal Operation Model

The operation model can be represented formally by extended finite automata (EFA) [21], a generalization of automata in-

Fig. 1. An operation in the SOP language.
including guards and actions. The guards and actions are associated with the transitions in the automaton. A transition in an EFA is enabled if and only if its corresponding guard formula is evaluated to be true; when the transition is undertaken, a set of variables may subsequently be updated. Formal tools for EFAs are available; see [15], [24], and [26]. Hence, both formal verification and synthesis can be applied directly in the suggested EFA models.

Definition 1 (Extended Finite Automaton): An extended finite automaton is a 6-tuple

\[
E = (Q \times V, \Sigma, G, A, \rightarrow, (q_0, v_0)).
\]

The set \( Q \times V \) is the extended finite set of states, where \( Q \) is a finite set of locations and \( V \) is the finite domain of definition of the variables. \( \Sigma \) is a nonempty finite set of events (the alphabet), \( G \) is a set of guard predicates over \( V \), \( A \) is a collection of action functions, \( \rightarrow \subseteq Q \times \Sigma \times G \times A \times Q \) is a state transition relation, and \((q_0, v_0) \in Q \times V \) is the initial state.

For notational purposes, the guards and actions are grouped into a set, \( C \), of transition conditions. These transition conditions include both the current and next values of the variables after a transition. The variable values after a transition (next values) are denoted \( v' \in V \). The conditions are a map \( V \times V \rightarrow \mathbb{B} \), \( v \in V, v' \in V \). Consider, for example, a guard, \((v_1 = 0) \land (v_2 = 1)\), combined with an action, \( v_1 := 1; v_2 := 0 \), which is expressed as a single transition condition, \((v_1 = 0) \land (v_1' = 1) \land (v_2' = 1) \land (v_2' = 0)\).

The following definition states how an operation can be modeled by an EFA.

Definition 2 (Operation): An operation can be described formally by an EFA where the set of locations \( Q_k = \{O_k^i, O_k^f, O_k^r\} \), the event set \( \Sigma_k = \{O_k^j\} \), the set of transition conditions \( C_k^t \), the transition relation \( \rightarrow_k = \{(O_k^i, O_k^j / C_k^t, O_k^f), (O_k^j / O_k^i / C_k^t, O_k^r)\} \), and the initial location \( q_0k = O_k^i \) (see Fig. 2).

First observe that the guards and actions in Definition 1 of an EFA are replaced in Definition 2 with the corresponding transition conditions. The transition for operation \( O_k \) from the initial location \( O_k^i \) to the execution location \( O_k^f \) is enabled when precondition \( C_k^t \) is satisfied, after which the transition can be fired and the event \( O_k^j \) occurs. In the same way, completion event \( O_k^j \) can only occur when postcondition \( C_k^t \) is fulfilled. For operations that need to be repeated, the operation model in Fig. 2 is modified. A reset transition is then introduced, from \( O_k^r \) to \( O_k^i \), such that operation \( O_k \) can be repeated from its initial location, \( O_k^i \), when a reset event, \( O_k^r \), has been fired. This can happen when reset condition \( C_k^r \) is satisfied. Resettable operations are not considered in this paper.

### Table I

<table>
<thead>
<tr>
<th>SOP</th>
<th>Sequences of Operations</th>
<th>( SO_k )</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFA</td>
<td>Extended Finite Automaton</td>
<td>( O_k )</td>
</tr>
<tr>
<td>( O_k^i )</td>
<td>Initial location</td>
<td>( O_k^r )</td>
</tr>
<tr>
<td>( O_k^f )</td>
<td>Finish location</td>
<td>( C_k^t )</td>
</tr>
<tr>
<td>( O_k^r )</td>
<td>Operation postcondition</td>
<td>( O_k^i )</td>
</tr>
<tr>
<td>( O_k^j )</td>
<td>Operation stop event</td>
<td>( O_k^r )</td>
</tr>
</tbody>
</table>

B. Condition-Based Operation Relations

The conditions are logical expressions implying on variable set \( V \). The operation will interact with the outside by reading and changing the values of these variables, which are defined by resources, products, and other operations. The conditions include both the current and the next values of the variables after the operation state.

To include requirements implying on the operation locations in the pre- and postconditions, these locations are also represented as Boolean variables \( O_k^i, O_k^f, \) and \( O_k^r \) included in variable set \( V \). Each of these variables, called operation variables, is equal to one when the corresponding location is active.

If an operation includes an operation variable in its transition condition, it is defined as directly related to that operation. If the precondition of \( O_k \) is, for example, \( C_k^t = O_k^i \lor O_k^f \), then \( O_k^t \) is directly related to the operations \( O_k \) and \( O_k^j \). Operation \( O_k \) is also directly related to other operations that use \( O_k^i, O_k^f, \) or \( O_k^r \) in their transition conditions.

Every operation is represented by this three-state model and is described as self-contained, since the relations among operations are stored in each operation. These relations are extracted from the operations when the sequences are visualized in the SOP language. This gives a flexible modeling language for complex systems, a language in which operations can be grouped into various sequences.

C. Sequences of Operations, SOPs

Each operation will start in its initial location and wait for its precondition to be fulfilled. If there are no preconditions, all operations will execute unrelated to each other. In practice, a number of relations between the operations restrict the behavior. For example, operation preconditions can be generated based on specific product assembly requirements [3]. To formally model the execution of the operations, the full synchronous composition operator \( \parallel \), defined for EFA by Sköldstam et al. [22] is used. For \( n \) operations, \( O_1, O_2, \ldots, O_n \), the composite operation model is \( O_1 \parallel O_2 \parallel \cdots \parallel O_n \). A SOP is defined as follows.

Definition 3 (SOP): SOP \( SO_k \) is defined by the composite model \( SO_k = O_k^1 \parallel O_k^2 \parallel \cdots \parallel O_k^n \), all restrictions being represented by pre- and postconditions included in the individual operation models. The operations involved in \( SO_k \) are included in the set \( O_k = \{O_k^1, \ldots, O_k^n\} \).

An operation \( O_{k_2} \) can be included in other SOPs as well, and is not dependent on any graphical model. Thus, it can also be grouped and visualized from various perspectives. A SOP can be represented graphically by a set of sequences, \( S_k = \{S_{k_1}, \ldots, S_{k_n}\} \), a sequence being defined as follows.
Definition 4 (Sequence): Sequence \( S_{ki} \) is a graph that connects a set of operations or groups of operations related to each other. The operation conditions in the sequence are represented by arrows, lines, and Boolean expressions.

Observe that here the sequence represents not only strictly sequential operation relations, but can also include groups of operations. The SOP language can group operations into parallel, alternative, arbitrary order, and hierarchical groups.

1) Example 1 (The SOP Language): Fig. 3 shows the four SOPs, \( SO_1, SO_2, SO_3, \) and \( SO_4 \); these present operations \( Q = \{O_3, O_8, O_9, O_10\} \) (see Table II), which are executed by the manufacturing cell shown in Fig. 4. The cell assembles a product with two parts and consists of a conveyor (1), a fixture (2), three robots (3,4,5), and an automated guided vehicle (6). Henceforth, the activities of the fixture and the work of the three robots will be examined.

The first SOP, \( SO_1 = \{O_1||O_2||\ldots||O_{10}\} \), includes two sequences. The left sequence describes operations realized by the fixture, and the right sequence operations performed by Robot 3. Operation \( O_1 \) fixes the product by means of two suboperations, \( O_2 \) and \( O_3 \), denoted by box \( b \). \( O_2 \) places the product in the fixture; as it is realized by both the robot and the fixture, it is included in both the left and the right sequences.

Operation \( O_3 \) closes the clamps of the fixture, which can only happen after \( O_2 \) has finished, i.e., a sequential relations indicated by the arrow \( a \). \( O_3 \) has an extra postcondition, \( p' = type \), which is an action that assigns variable \( p \) the value of the sensor \( type \) when \( O_3 \) occurs. This sensor checks what type of product is located in the fixture and decides whether to execute operation \( O_8 \) or \( O_9 \) later on.

The parallel lines at \( c \) represent a parallel relation, meaning that both \( O_4 \) and \( O_5 \) are subject to \( O_4^f \) as a precondition. Since \( O_4 \) has the extra precondition \( O_4^f \), the complete precondition for \( O_6 \) is \( C_6^c = O_4^f \land O_3^{r_1} \). Operation \( O_6 \) will release the product from the fixture, which can be executed in parallel with \( O_4 \) and \( O_5 \). Operations \( O_4 \) and \( O_5 \) write production information to a logging system.

The operations realized by Robot 3 are shown in the right sequence of \( SO_1 \) in Fig. 3 starting with \( O_2 \), which places part \( B \) in the fixture. After that, \( O_7 \) changes the robot’s tool from a gripper to a drill. Based on the type of product located in the fixture, either drilling operation \( O_8 \) or \( O_9 \) is executed. This alternative branch is shown at \( d \) and is modeled by adding an alternative booking variable to each operation. Operation \( O_{10} \) rivets the last holes shared by the two product types, but after \( O_{12} \) and \( O_{13} \), which are realized by other robots shown in \( SO_4 \). The final precondition for \( O_{10} \) is \( C_{10} = O_4^f \land O_3^{r_1} \land (O_8^f \lor O_9^f) \).

As can be seen in Fig. 3, it is not obvious when graphic connections or Boolean expressions should be used to describe the complex operation conditions. For example, if every operation in a normal industrial manufacturing cell is included in a single sequence, it is usually very complicated to understand the operation relations. Even creating that sequence can sometimes be almost impossible. A challenge when representing relations among operations is therefore to decide which operations to include in a given sequence.

D. Operation Relations in Multiple SOPs

SOP \( SO_1 \) in Fig. 3 includes operations \( O_1, \ldots, O_{10} \). These can be graphically represented in many ways; in the figure they are structured in two sequences, one for each resource. An operation can have multiple attributes defining its properties, and these attributes determine how the operation is structured. An attribute can be related to product information, executing resources, safety constraints, etc. These attributes are used to extract operations into SOPs and to graphically structure them in different sequences and SOPs.

1) Example 2 (Structure Operations): Let us go back to the cell in Fig. 4 and consider operations \( O_{11} \) to \( O_{16} \), which according to \( SO_4 \) in Fig. 3 are related to Robots 4 and 5. The
product sequence is represented by $SO_2$, which includes only operations directly involved in assembling the product (see [3]). The product consists of two parts that are placed and fixed in the fixture by $O_{11}$ followed by $O_1$, processed by $O_{16}$, released by $O_6$, and finally moved away by $O_{15}$. This sequence is based on the product requirements and design and can be described in early design phases.

The operations in $SO_2$ are assigned to different resources during the layout design of the cell. The process operation is complicated since it involves two robots. Therefore, the suboperations of $O_{16}$ are shown in a separate SOP, $SO_3$. The SOP language allows these suboperations to be included in $SO_2$, but in this case are not interesting to the designer of $SO_2$.

Programming the various control systems calls for more detailed operations that describe every task. However, this is complicated in the case of operations structured as in $SO_2$. Instead a structure based on their realizing resources, shown in $SO_1$ and $SO_4$, could be used, in which the operations are structured in sequences based on each resource. The relations between the resources are shown using Boolean expressions, for example, indicating that operation $O_{10}$ must wait for the completion of $O_{12}$ and $O_{13}$. These added Boolean expressions convey a clear understanding of the operation sequence for each resource. Most of the operations shown in Table II also include suboperations, for example, $O_{13}$, which adds multiple rivets, but is not defined at this stage.

It is very complicated to understand and manually organize all operations in a real development project since hundreds of operations may be involved. One aid could be to automatically create a set of sequences that represents a SOP. For example, if the control designer is interested in the interaction between the robots and the fixture, involving $O_2$, $O_3$, $O_{10}$, $O_{12}$, and $O_{15}$, the SOP

$$SO_2 = O_2||O_{10}||O_{12}||O_{15}$$

would be interesting to visualize graphically. This is done by first identifying the relations among the operations, as discussed in the next section, and then visualizing the identified relations in sequences, as presented in Section IV.

## III. Relation Identification

The direct relations between operations stem from various design requirements and decisions, such as product assembly or manufacturing safety constraints. However, most operations will not be directly related to each other, even though they are related in some way. For example, if we look at operations $O_{13}$, $O_{14}$, and $O_{15}$ from Fig. 3, with preconditions $C_{14} = O_{13}$ and $C_{15} = O_{14} \land O_{16}$, i.e., a sequence, then $O_{13}$ and $O_{15}$ are not directly related but $O_{13}$ will always precede $O_{15}$. They are thus indirectly related.

To use multiple SOPs, it is crucial to identify these indirect relations among operations. Therefore, let us examine the types of relations that exist between operations and how to identify them.

### A. Operation Relation Types

To analyze and reason about the relations between operations, the possible locations of an operation, related to when an event is enabled, will be studied. If we have a set of operations $Q = \{O_1, \ldots, O_n\}$ that defines the complete behavior of a system, the composition model $H = O_1 \land \cdots \land O_n$ defines the relations among them. The finite set of locations of $H$ is denoted $Q_H$.

Operation $O_k$ will be located in one of its three locations, $q_k \in Q_k$, in every composition location $q_H \in Q_H \subseteq Q_1 \times \cdots \times Q_n$, according to the synchronization operator. The composition location $q_H$ therefore includes the current location of each individual operation in $H$, i.e., $q_H = \langle q_1, \ldots, q_n \rangle$. The events that are enabled in location $q_H$ are denoted as $\Gamma^Q(q_H)$, and two location sets related to an event $\sigma$ can be defined as follows:

- $Q_H^\sigma : \{q_H \in Q_H^\sigma \in \Gamma^Q(q_H)\}$—the locations in $Q_H$, where $\sigma$ is enabled;
- $O_k^\sigma : \{q_k \in Q_k | q_H = \langle q_1, \ldots, q_k, \ldots, q_n \rangle \in Q_H^\sigma\}$—the locations of operation $O_k$ where $\sigma$ is enabled.

There are seven possible location combinations of $O_k^\sigma$, i.e., $\{O_k^\sigma\}$, $\{O_k^\sigma\}$, $\{O_k^\sigma\}$, $\{O_k^\sigma\}$, $\{O_k^\sigma\}$, $\{O_k^\sigma\}$, and $\{O_k^\sigma\}$. For example, if $O_k^\sigma = \{O_k^\sigma\}$, then operation $O_k$ will only start when operation $O_k$ is in its final location. To define the possible relations between operations $O_k$ and $O_i$, all four location sets, i.e., $O_k^\sigma$, $O_k^\sigma$, $O_k^\sigma$, and $O_k^\sigma$, must be identified and compared. The possible combinations of these state sets can be grouped into the following relation types.

**Definition 5 (Relations Between $O_k$ and $O_i$):**

- Always in sequence: $O_k \supset O_i$
  
  $$O_k \supset O_i : O_k^\sigma = \{O_k^\sigma\} \land O_i^\sigma = \{O_i^\sigma\} \land \emptyset$$

- Sometimes in sequence: $O_k \supseteq O_i$
  
  $$O_k \supseteq O_i : O_k^\sigma = \{O_k^\sigma\} \land O_i^\sigma = \{O_i^\sigma\} \land \emptyset$$

- Parallel: $O_k \parallel O_i$
  
  $$O_k \parallel O_i : O_k^\sigma = \{O_k^\sigma\} \land O_i^\sigma = \{O_i^\sigma\} \land \emptyset$$

- Alternative: $O_k + O_i$
  
  $$O_k + O_i : O_k^\sigma = \{O_k^\sigma\} \land O_i^\sigma = \{O_i^\sigma\} \land \emptyset$$

- Arbitrary order: $O_k \oplus O_i$
  
  $$O_k \oplus O_i : O_k^\sigma = \{O_k^\sigma\} \land O_i^\sigma = \{O_i^\sigma\} \land \emptyset$$

- Hierarchy: $O_k \subseteq O_i$
  
  $$O_k \subseteq O_i : O_k^\sigma = \{O_k^\sigma\} \land O_i^\sigma = \{O_i^\sigma\} \land \emptyset$$
is in its initial location. This is the normal sequential relation in which two operations are executed one after the other.

Sometimes in sequence, \( O_k \preceq O_{\ell} \), indicates that \( O_k \) sometimes precedes \( O_{\ell} \). This relation is almost as always in sequence, in which \( O_k \) will only start when \( O_{\ell} \) is in its initial location, but \( O_{\ell} \) can start when \( O_k \) is in either its final or initial location. Consider Fig. 5, which shows an automaton and a SOP with a sometimes in sequence relation between \( O_k \) and \( O_{\ell} \), and \( O_k \) = \( O_{\ell} \) \( \lor \) \( O_{k}^f \). If \( O_k \) is executed, then \( O_k^f \) is always enabled when \( O_{\ell} \) is in its initial location \( O_{\ell}^i \), and \( O_k^f \) is enabled when \( O_k \) is located in \( O_{\ell}^i \). However, when \( O_{\ell} \) is executed instead of \( O_k \), \( O_{\ell}^f \) is also enabled when \( O_k \) is in \( O_{\ell}^i \), i.e., \( O_k^f = \{ O_k^i, O_{\ell}^f \}. \)

Parallel, \( O_k || O_{\ell} \), indicates that \( O_k \) and \( O_{\ell} \) can execute in parallel, i.e., any combination of events exists in the complete behavior of a system.

Alternative, \( O_k \oplus O_{\ell} \), indicates that if \( O_k \) or \( O_{\ell} \) is not located in the initial location, the other one cannot start. This relation is usually due to an alternative branch, modeled by the booking of a mutually exclusive resource.

Arbitrary order, \( O_k \ominus O_{\ell} \), indicates that \( O_k \) and \( O_{\ell} \) will not execute at the same time, but does not specify the order in which they execute. This normally happens when two operations require the same resource, which hinders them from executing simultaneously. This means that once an operation has started, it must complete before the other operation can start, but the order in which the operations are executed is not specified.

Hierarchy, \( O_k \sqsubset O_{\ell} \), indicates that \( O_k \) is a parent of \( O_{\ell} \). This relation implies that \( O_{\ell} \) starts and completes when \( O_k \) is located in its executing location.

Other, \( O_k \land O_{\ell} \), indicates that \( O_k \) is related to \( O_{\ell} \) in some way, i.e., all other possible location combinations. Obviously, these relations are interesting, but are not currently handled by the visualization algorithms introduced in Section IV. New operation relations will be defined and managed in future research.

Observe that all four location sets, i.e., \( O_k^g \), \( O_k^h \), \( O_k^f \), and \( O_k^i \), must be considered when defining the relations between two operations. Consider, for example, the definition of arbitrary order in Definition 5. Operation \( O_{\ell} \) can start when \( O_k \) is in either its initial or final location and stop when \( O_k \) is in either its initial or final location. However, \( O_{\ell} \) cannot start in \( O_k^f \) and stop in \( O_k^i \), since \( O_k \) does not start or stop in the execution location of \( O_{\ell} \), according to \( O_k^{\bullet} = \{ O_k^i, O_k^f \} \) and \( O_{\ell}^{\bullet} = \{ O_{\ell}^i, O_{\ell}^f \}. \) Hence, when one operation starts, it must be completed before the other operation starts.

B. Identifying Relations

To identify the relations among a set of operations \( O_{\beta} = \{ O_1, \ldots, O_{\ell} \}, \) where \( O_{\beta} \subseteq O = \{ O_1, \ldots, O_n \} \) and \( O \) includes every operation of the system, Algorithm 1 is used.

Algorithm 1: Identify operation relations

Input: \( O_{\beta}, O \)

Result: RelationTable

\[ H = \text{Sync}(O) \]

foreach \( O \) in \( O_{\beta} \) do

\[ Q_H^O = \text{FindEnabled}(O^1, H); \]

\[ Q_H^O = \text{FindEnabled}(O^1, H); \]

end

\[ O^* = \text{ExtractOperations}(Q_H^O); \]

foreach \( O_s \) in \( O^* \) do

foreach \( O_t \) in \( O^* \) do

RelationTable \([O_s, O_t] = \text{MatchRel}(O_s, O_t); \]

end

end

Algorithm 1 takes \( O_{\beta} \) and \( O \) as input and returns a table describing the pairwise relation between the operations. The first step of the algorithm is to synchronize the operation models with the full synchronous composition operator and store the result in \( H \). Then, the location set \( Q_{H}^O \) is calculated including the elements \( O_{s}^{O_f} \) and \( O_{t}^{O_f} \), where \( s = k \ldots \ell \), by the first FOR loop. After that, the operations are extracted into \( O^* \) and iterated through and compared pairwise according to Definition 5, i.e., the location sets \( O_{s}^{O_f} \), \( O_{s}^{O_f} \), \( O_{s}^{O_f} \), and \( O_{t}^{O_f} \), where \( s = k \ldots \ell \), \( t = k \ldots \ell \), and \( s \neq t \), are created and the relations are matched.

In the following example, the relations among the operations in SOP \( SO_5 \) in (1) will be identified.

1) Example 3 (Relation Identification): Consider SOP \( SO_5 \) with the operations \( O_5 = \{ O_2, O_3, O_{10}, O_{12}, O_{15} \} \). First, the complete system \( H = O_1 \ldots \} \) is generated, the locations in \( H \) being built of the location of every operation, for example, the initial location is named \( \{ O_{1}^{\bullet}, \ldots, O_{16}^{\bullet} \}. \)

After that, the location sets \( Q_H^{O_1} \) and \( Q_H^{O_2} \), where \( s = 2, 3, 10, 12, 15 \), are calculated, i.e., the locations in which the start and stop events of the operations in \( O_5 \) are enabled, in the complete system \( H \). This is actually done simultaneously with the synchronization to avoid iterating the locations once again. Two of the location sets are

\[ Q_H^{O_2} = \{ O_1^i, O_2^i, O_3^i, \ldots, O_{15}^i \}, \{ O_1^f, O_2^f, O_3^f, \ldots, O_{15}^f \}; \]

\[ Q_H^{O_2} = \{ O_1^i, O_2^i, O_3^i, \ldots, O_{15}^i \}, \{ O_1^f, O_2^f, O_3^f, \ldots, O_{15}^f \}; \]

Since we are interested in only the relations among the operations in \( O_5 \), the above sets are reorganized based on when the event is enabled relative to the locations of the operations in \( O_5 \).
The location sets $O_2^{O_1}$ and $O_s^{O_1}$ are the result. Below, only the sets for the start event of each operation are shown:

\[
\begin{align*}
O_2^{O_1} & : \{O_2^{O_1}\}, \{O_3^{O_1}\}, \{O_{10}^{O_1}\}, \{O_{12}^{O_1}, O_{12}^{O_1}, O_{12}^{O_1}\}, \{O_{15}^{O_1}\} \\
O_3^{O_1} & : \{O_2^{O_1}\}, \{O_3^{O_1}\}, \{O_{10}^{O_1}\}, \{O_{12}^{O_1}, O_{12}^{O_1}, O_{12}^{O_1}\}, \{O_{15}^{O_1}\} \\
O_{10}^{O_1} & : \{O_2^{O_1}\}, \{O_3^{O_1}\}, \{O_{10}^{O_1}\}, \{O_{12}^{O_1}\} \\
O_{12}^{O_1} & : \{O_2^{O_1}, O_2^{O_1}, O_2^{O_1}\}, \{O_3^{O_1}, O_3^{O_1}, O_3^{O_1}\}, \{O_{10}^{O_1}, O_{12}^{O_1}, O_{12}^{O_1}\}, \{O_{15}^{O_1}\} \\
O_{15}^{O_1} & : \{O_2^{O_1}, O_3^{O_1}, O_{10}^{O_1}, O_{12}^{O_1}, O_{15}^{O_1}\}.
\end{align*}
\]

These relations are based on both direct and indirect relations. \hfill \square

C. Computation

When the relations within only a small subset of operations in a system are identified, all the operations in the system must still be synchronized in any case. Since these computations usually involve large state set analysis, state-space explosion can be an issue in real systems. This can be handled using binary decision diagrams (BDDs) [1], as in Vahidi [25] and Miremadi et al. [15]. Recent results [20] indicate that a controller can be synchronized and synthesized based on EFA operation models and on BDDs for systems with $10^{11}$ reachable states using the software tool Supremica [24]. The implementation details are omitted here due to space constraints.

When all operations have been synchronized, pairwise identification will be done only on the subset of operations to be visualized. The maximum number of comparisons is $(n(n-1)/2)$, where $n$ is the number of operations to be visualized. The proof-of-concept implementation is promising for real industrial systems, but further investigation is needed, especially concerning how to recompute the relations when only some of the operations have been changed.

IV. RELATION VISUALIZATION

The relations among operations in a SOP can be visualized in many ways, since the relations can be presented both graphically and using Boolean expressions. For example, based on operation attributes such as resource names and product parts, it is possible to visualize the operations in different sequences, such as in $SO_1$ and $SO_2$ in Fig. 3. However, it is also possible to create only one sequence including all the operations in the SOP.

The algorithm presented in this section takes a set of operations and creates a sequence including all operations in the given set. If the operations in the SOP are to be organized based on an operation attribute, the operations must be structured according to the attributes before the algorithm can create the sequences.

A. Grouping

The visualization algorithm is divided into four stages: relation identification, grouping, sequencing, and drawing. The relations within a set of operations are first identified as in Section III. After that, the operations are grouped based on hierarchy, alternative, arbitrary order, or parallel relations, called related relations. After the operations have been grouped, they are sorted based on whether they are predecessors or successors (always and sometimes in sequence). Finally, the sequence graph is drawn. The other relations are not handled by the algorithm (see the end of this section).

To sort the operations into groups and sequences, each operation and group becomes a node in a data structure. Group and operation nodes are almost the same, except that a group can contain other nodes and the relation between the group and other nodes is based on all operations included in the group (and its subgroups). For example, if a group precedes an operation, this means that all nodes in the group precede the operation.

An operation node for each operation is created and added to the list Nodes. An operation node always represents an operation and is therefore sometimes also referred to as an operation. The list Nodes is then used as input to Algorithm 2.

Algorithm 2: Grouping of operations

Input Nodes

Result /Create Hierarchy Groups groupList := new Node List;

foreach n in Nodes do

If n is parent then

\[
gH \ := \ \text{New Hierarchy group};
\]

foreach k in Nodes do

If k child of n then

\[
gH.add(k);
\]

end

end

groupList.add(gH);

end

Nodes := ResolveGroups(groupList, Nodes);

//Create Alternative Groups

foreach n in Nodes do

| {\ldots} |

end

Nodes := ResolveGroups(groupList, Nodes);

//Create Arbitrary Groups

foreach n in Nodes do

end

Nodes := ResolveGroups(groupList, Nodes);

//Create Parallel Groups

foreach n in Nodes do

end

Nodes := ResolveGroups(groupList, Nodes);

The grouping Algorithm 2 creates hierarchy, alternative, arbitrary order, and parallel groups. First, the hierarchies are iden-
tified and created, which checks each node in \( \text{Nodes} \) to see whether it is a parent (i.e., whether the operation has suboperations). When a parent is found, a new hierarchy group is created and the child nodes from \( \text{Nodes} \) are added. The command \( k \) child of \( n \) checks the previously identified operation relation, to determine whether \( k \) is a child of \( n \). Some of the created groups can contain the same nodes, since a parent can also be a parent to a child of a child. The groups are therefore resolved and merged if they are referencing the same nodes.

\( \text{ResolveGroups} \) is a recursive method that merges created groups if they share nodes. The set \( G = \{ G_1, \ldots, G_n \} \) represents the groups in \( \text{groupList} \), where \( G_i \) is a set of the nodes in a group. If \( G_i \cap G_j \neq \emptyset \), \( i \neq j \), a new group is created including the nodes of both groups, \( G_{\text{new}} = G_i \cup G_j \). New intersections with \( G_{\text{new}} \) and the remaining groups in \( G \) are resolved recursively. After that, new groups are created in the same way based on the groups not included in \( G_{\text{new}} \).

The result of \( \text{ResolveGroups} \) is a new \( \text{Nodes} \) list containing the created hierarchy groups and remaining nodes not included in a group. This list is then used when alternative groups are created and resolved. The method for creating alternative groups is the same, except that alternatives are also identified inside each hierarchy group. An alternative group includes nodes that share at least one alternative relation. Observe that not all nodes in the group must have an alternative relation with every other node in the group. Arbitrary order and parallel groups are created recursively in the same way as are the alternative groups.

1) Example 4 (Grouping): Consider the operations from the relation identification example. A \( \text{Nodes} \) list is created including these operations, shown in Fig. 6(a). There are no hierarchy, alternative, or arbitrary order relations, but rather there are parallel relations. First, each operation is studied; when it has parallel relations with other operations, a parallel group \( G_{P2} \) is created. In Fig. 6, three groups are created: \( G_{P2} \), \( G_{P3} \), and \( G_{P12} \). Operations \( O_{10} \) and \( O_{15} \) do not have any parallel relations.

As can be seen in Fig. 6, the three groups are related to each other since they share nodes. First, \( G_{P2} \cap G_{P3} \) = \( \{ O_{12} \} \) is calculated, i.e., they share node \( O_{12} \). The new group \( G_P = G_{P2} \cup G_{P3} \) is created, including nodes \( O_2, O_5 \), and \( O_{12} \) (b). The new \( G_P \) is tested with the remaining group, \( G_P \cap G_{12} = \{ O_2, O_3, O_{12} \} \), which results in \( G_P = G_P \cup G_{12} \). All groups are now resolved and included in \( G_P \).

The \( \text{ResolveGroups} \) method adds the new group to \( \text{Nodes} \) and removes the nodes that were included in the group from \( \text{Nodes} \). The result is \( \{ G_P, O_{10}, O_{15} \} \).

When all groups have been identified and created, the sequential relations need to be identified, which is handled by the sequencing algorithm.

B. Sequencing

To structure the elements in \( \text{Nodes} \) sequentially, each node \( n \) is given a set of attributes. Two attributes are references to the closest preceding node, \( n.Pred \), and the successor node, \( n.Succ \). These attributes are used to organize the nodes into a tree structure. Three other attributes are the lists \( n.PredList \), \( n.SuccList \), and \( n.RelList \) used to sort a set of nodes related to \( n \). The list from the grouping algorithm is the input to the recursive sequencing Algorithm 3.

Algorithm 3: Sequencing nodes

\( \text{Seq}(\text{Nodes}) \) return root Node
If \( \text{Nodes.Empty then return null} \);
root := \( \text{ExtractNode}(\text{Nodes}) \);
If \( \text{root is group then return Seq(root.nodes)} \);
foreach \( n \) in \( \text{Nodes} \) do
    If \( n \) is Pred to root then \( \text{root.PredList.add(n)} \);
    else If \( n \) is Succ to root then \( \text{root.SuccList.add(n)} \);
    else \( \text{root.RelList.add(n)} \);
end
root.Pred := GetLast(Seq(root.PredList));
root.Succ := Seq(root.SuccList);
Seq(root.RelList);
return Structure(root);

The recursive Seq method takes a list of nodes and returns the root node of a tree. The first line terminates the recursive algorithm. The next line extracts one of the nodes from the set \( \text{Nodes} \) that is used as root by the algorithm. If root is a group, the nodes inside it are also sorted by the Seq method.

Each remaining node in \( \text{Nodes} \) is sorted according to whether it is a predecessor of, successor to or only related to root and added to root.PredList, root.SuccList or root.RelList.

The nodes in the lists of root are then sorted by Seq. This recursive method terminates when a leaf is reached, meaning that a list is empty. The returning node from the sorting of PredList and SuccList is first in a subsequence, but the predecessor to root is the last node in the subsequence the GetLast method finds it and assigned it to root.Pred. The returning node is assigned to root.Succ, hence, a tree is constructed with root as the first node.

The first node in the sequence can be found when the root.Pred branch is followed. To better represent the real sequence, the tree is restructured based on the sequence order, i.e., only the SuccList branch exists. For example, if \( A \) is root and \( A.Pred = B \) and \( A.Succ = C \), then \( B \) replaces \( A \) as root and \( B.Succ = A \) and \( A.Succ = C(A.Pred = Null) \), i.e., the tree is restructured with the first node in the sequence at the top.

1) Example 5 (Sequencing): In Fig. 7, the list from grouping example (a) is input to the sequencing algorithm. \( O_{10} \) is picked as root and the other nodes are sorted into \( O_{10}.PredList \) and \( O_{10}.SuccList \), represented by the \( p \) and \( s \) arrows in (b). Since
only $O_{15}$ is in $O_{10}.\text{SuccList}$, that branch is resolved, i.e., $O_{10}.\text{Succ} = O_{15}$. The node in $O_{10}.\text{PredList}$ is a group, and its included nodes are sorted in the same way: $O_3$ is picked as root, $O_2$ precedes $O_5$ and is added to $O_3.\text{PredList}$, and $O_{12}$ has a parallel relation with $O_2$ and is therefore not sorted. Finally, in (d), the nodes are restructured based on the sequence order. $G_P$ is returned as root node.

Once the node tree has been created, it is straightforward to generate the resulting sequence graph.

C. Drawing

When drawing the sequence, each node in the tree is drawn, one at a time: first the root node is drawn, after that the successor node, which is connected to the root node, and then the node after that until the last node has been drawn. If the node is a group, the nodes in the group are drawn inside the group type notation (cf. the parallel group $G_P$ in Fig. 8, where the parallel lines are drawn above and below the included operations).

The lines between the operations and groups that define the sequential relations are drawn differently if the relation is direct, i.e., is defined in the operation condition, or indirect. If the relation is indirect, a dashed line is used instead.

1) Example 6 (Drawing): SOP $SO_5$, based on the sorting and grouping example above, is shown in Fig. 8, including one sequence. All of the relations are indirect except between $O_2$ and $O_3$, since $C^I_3 = O^I_3$.

One limitation of the presented algorithm is that operations that have an other relation with at least one operation in the sequence cannot be included. These operations are placed next to the sequence showing their conditions as Boolean expressions. Another limitation is that no loops can be included, since they are not explicitly identified in the relation identification. These limitations are an area for future research.

D. Visualizing a SOP

Methods for identifying and visualizing operations in one sequence have now been presented. However, one of the main features of the SOP language is that it lets one present the SOP operations in multiple sequences. It is usually complicated or even impossible to represent all operation relations in only one sequence. In Fig. 9, a SOP with four sequences is visualized, the relations between the sequences being presented using Boolean expressions.

When visualizing multiple sequences, the relations among the SOP operations are first identified. After that, each sequence is sorted and visualized individually. When the sequences are drawn, all operation relations that are not represented by graphical notations in any of the sequences are then described by Boolean pre- and postconditions. For example, the precondition of operation $O_F$ in Fig. 9 is $C^I_F \equiv O^I_E \land O^I_J \land O^I_B$, where $O^I_E$ and $O^I_J$ are represented graphically in the second and fourth sequences and $O^I_B$ is described by the Boolean expression.

The next section presents a software tool called SP [16] that visualizes and manages operations and their relations, and coordinates changes and updates among the operations and the SOPs.

V. SEQUENCE PLANNER (SP)

Sequence Planner (SP) is a prototype software tool developed to manage the SOP language and all involved operations. A screen shot is shown in Fig. 10. SP has a central repository storing all operations used in a project, shown in the upper left in the figure. In addition, a resource structure and a product structure are available. The operations are stored separately from the SOPs and the graphical representation.

SP allows the creation of any number of SOPs, one of which is shown in Fig. 10. A SOP can be created, for example, by clicking on a resource, showing all operations realized by that resource, or manually by dragging operations into the SOP window. In a SOP, new conditions on the operations can be added by either creating sequences or adding Boolean
expressions. Changes related to the operations and SOPs can also come from external applications such as PLM, simulation, verification, optimization, or synthesis tools, or can be manually changed in SP. Operation conditions may change, new operations be added, or product and/or resource data be updated. One challenge in SP is therefore to coordinate changes among the SOPs and the operations.

A. Updating Operations in Various SOPs

Every self-contained operation is stored in SP in set \( \mathcal{O} \), which is unrelated to the SOPs. \( \mathcal{O} \) is the main operation set for the whole model and is the interface with external applications, not only receiving changes but also sending out operation information. An operation is not fully included in the project until it has been included in \( \mathcal{O} \).

It is possible to create new operations, or extract operations from \( \mathcal{O} \), when creating a new SOP. However, when updating and managing changes between the SOPs and \( \mathcal{O} \), it is not feasible for each SOP to directly change the operations in \( \mathcal{O} \). Therefore, the SOP uses copies of the operations, and when changes are made in one SOP, they are saved to \( \mathcal{O} \) and coordinated with the other SOPs.

Since the same operation can exist in multiple copies, an operation object is used to keep them apart, defined as follows.

**Definition 6 (Operation Object):** Operation object \( o_k \) is a data object representing operation \( O_k \). Object \( o_k \) has two important attributes: the id of the operation, \( o_k, id = k \), and the operation information-like conditions and realizing resources, stored in \( o_k, value \).

If two operation objects \( o_s \) and \( o_t \), where \( o_s \neq o_t \), have the same id, \( o_s, id = o_t, id = k \), they represent the same operation, i.e., \( O_k \). The different operation objects, however, can end up with differing values, which must be managed by SP.

Each operation in the SOPs and in \( \mathcal{O} \) is represented by an operation object. For simplicity, in the rest of this section, \( \mathcal{O} \) represents all operation objects and \( O_k \) represents operation objects for SOP \( SO_k \).

When an operation object is changed in a SOP or in \( \mathcal{O} \), other operation objects representing the same operation have differing values and need to be updated. To identify differing operation objects in set \( \mathcal{O} \) compared with set \( \mathcal{O}_t \), the set \( \Delta(\mathcal{O}_s, \mathcal{O}_t) \) is introduced, which is defined as follows.

**Definition 7 (Differing Operation Objects):** The differing operation object set, \( \Delta(\mathcal{O}_s, \mathcal{O}_t) \), is defined as the set of operation objects \( o_k \in \mathcal{O}_s \), such that \( \exists o_t \in \mathcal{O}_t | o_k, id = o_t, id \wedge o_k, value \neq o_t, value \).

It is also interesting to identify operations included only in set \( \mathcal{O}_s \) but not in \( \mathcal{O} \). The new operation object set is defined as follows.

**Definition 8 (New Operation Objects):** The new operation object set, \( \mathcal{N}(\mathcal{O}_s, \mathcal{O}) \), is defined as the set of operation objects \( o_s \in \mathcal{O}_s \), such that \( \forall o_t \in \mathcal{O} | o_s, id \neq o_t, id \wedge s \neq t \).

When an operation object is changed in a SOP, it will first be saved to \( \mathcal{O} \). After that, the other SOPs are notified of the change and will update the changed operation and related sequences. No new changes can be added to \( \mathcal{O} \) in this period. This can be formulated in the following algorithmic steps.

1) When the changes made to \( SO_k \) are saved, \( \Delta(\mathcal{O}_s, \mathcal{O}) \) and \( \mathcal{N}(\mathcal{O}_s, \mathcal{O}) \) are calculated. The differing operation objects in \( \Delta(\mathcal{O}_s, \mathcal{O}) \) replace the old versions in \( \mathcal{O} \). The new operation objects, \( \mathcal{N}(\mathcal{O}_s, \mathcal{O}) \), are added to \( \mathcal{O} \).
2) Every other view \( SO_k \) where \( \Delta(\mathcal{O}_s, \mathcal{O}) \neq \emptyset \) creates an updated version of \( SO_k \).

When new operation relations requirements are introduced in one SOP, the impact of these changes could be better understood in other views. For example, if a product SOP is created, the work of building, for example, safety or resource requirements might result in changes to the sequences in the product SOP, leading to a need to change how the product is built. Completely understanding complex operation information and how changes affect system behavior calls for viewing the operations using multiple and coordinated SOPs.

VI. CONCLUSION

The relations among operations tend to be highly complex when developing an automation system. These relations are constantly changing throughout the development process, which makes it difficult to keep them up to date. Therefore, it would be more natural to consider sequence planning as an iterative process, in which sequences are viewed based on current requirements, rather than as a manual construction process. Using a graphic, formal language called Sequences of Operations (SOPs), this paper presents algorithms and methods that visualize the operation relations from multiple perspectives.

A set of sequences is used to structure the operations, for example, based on the resources used to realize them or based on how different parts are manufactured in the system. The structuring can be automated by using different user-assigned operation attributes, such as what resource was used to realize the operation. When the operations are structured, the relations among them are identified. This is accomplished by modeling the operations using EFA, which are synchronized in order to identify when an operation can execute relative to other operations. Based on the identified operation relations, a set of sequences can be created with a set of algorithms that visualizes the relations in the SOP language.

The proposed algorithms are implemented in a software tool called Sequence Planner, which uses the tool Supremica for automata calculations. Initial results and case studies indicate that the proposed approach is promising from both the computational and usability points of view. Further development is needed, focusing especially on how to recompute the relations when only some of the operations have changed.

An industrial case study is currently being conducted in which the approach is being evaluated during the design and installation of a complete manufacturing system. In addition, how to adapt Sequence Planner to other virtual process planning tools is being evaluated.

REFERENCES


Knut Åkesson received the M.S. degree in computer science and engineering from Lund Institute of Technology, Lund, Sweden, in 1997, and the Ph.D. degree in control engineering from Chalmers University of Technology, Gothenburg, Sweden, in 2002.

Currently, he is an Associate Professor at the Department of Signals and Systems, Chalmers University of Technology, where his main research interest is to develop and applying formal methods for verification and synthesis of control logic.

Chengyin Yuan received the B.S. and M.S. degrees in mechanical engineering from Tsinghua University, Beijing, China, and the Ph.D. degree in manufacturing automation controls from the University of Illinois at Urbana–Champaign, Urbana, in 1997, 1999, and 2004, respectively.

From 2005 and 2010, he was with the General Motors Research and Development Center, Warren, MI, where he was a Senior Researcher in the Manufacturing Systems Research Laboratory. Since 2010, he has been with Pride Power System Technology Co., Ltd., Beijing, China. His current research interests include virtual manufacturing, math-based automation system design and validation, distributed systems and simulation, and reconfigurable automation controls.

Sajed Miremadi was born in 1983 in Linköping, Sweden. He received the B.Sc. degree in computer engineering from Sharif University of Technology, Tehran, Iran, in 2006 and the M.Sc. degree in computer science from Linköping University, Linköping, Sweden, in 2008. He has been working towards the Ph.D. degree in automation at Chalmers University of Technology, Gothenburg, Sweden, since 2008.

His current research interests include supervisory control and optimization of (timed) discrete event systems, using formal methods.

Petter Falkman received the M.Sc. degree in automation and the Ph.D. degree in electrical engineering both from Chalmers University of Technology, Gothenburg, Sweden, in 1999 and 2005, respectively.

He is currently a Senior Lecturer at the Department of Signals and Systems, Chalmers University of Technology, where he is also part of the Wingquist Laboratory, a research center for virtual product and production development. Since 2009, he has been Program Director for the Automation and Mechatronic Program at Chalmers University of Technology. His main research topic concerns virtual preparation, specification and optimization of discrete event systems for production systems. This research is performed in collaboration with and support from a number of companies including ABB, Dassault Systemes, Siemens, and Volvo.