Synthesising process controllers from formal models of transformable assembly systems

Synthesising Process Controllers from Formal Models of Transformable Assembly Systems[†]

Lavindra de Silva [a,*], Paolo Felli [b], David Sanderson [a], Jack C. Chaplin [a], Brian Logan [c], and Svetan Ratchev [a]

[a] *Institute for Advanced Manufacturing, The University of Nottingham, University Park, Nottingham, NG7 2RD, UK*
[b] *Faculty of Computer Science, Free University of Bozen-Bolzano, Bolzano, 39100, Italy*
[c] *School of Computer Science, The University of Nottingham, Jubilee Campus, Nottingham, NG8 1BB, UK*
[*] *Corresponding author. Tel.: +44-(0)115-748-6352; fax: +44-(0)115-951-3666.*
E-mail addresses: Lavindra.deSilva@{nottingham.ac.uk,eng.cam.ac.uk}

Abstract

When producing complex and highly customisable products in low volumes (or in 'batch sizes of one'), automation of production systems is critical for competitiveness and profitability in high labour-cost economies. To facilitate batch-size-of-one production, 'topology generation', 'realisability', and 'control' algorithms have been developed as part of the Evolvable Assembly Systems (EAS) project. The topology generation algorithm computes all the possible sequences of parallel activities that assembly resources can perform on parts and is run offline whenever the layout of the production facility changes, whereas realisability checking and controller generation are performed at run-time to check *whether* a production facility with a given set of assembly resources can assemble a desired product, and *how* the product should be assembled, e.g., which resources to use, and when. Generated controllers are output in Business to Manufacturing Markup Language (B2MML). Taken together, the algorithms thus represent a step toward a complete path from the formal specification of an assembly system and the products to be assembled, to the automated synthesis of executable process plans. This paper presents each algorithm in sufficient detail to allow their reimplementation by other researchers. Topology generation is the most expensive step in the approach. A preliminary experimental evaluation of the scalability of topology generation is presented, which suggests that, for small to medium sized production facilities, the time required for recomputing the topology is sufficiently small not to preclude frequent factory transformations, e.g., the addition of new resources.

Keywords: Evolvable Assembly Systems; Controller Synthesis; Standardisation.

[†]This paper extends and subsumes the work published in [11].

1 Introduction

Advanced manufacturing is the result of the trend towards the integration of informatics with traditional manufacturing and assembly systems. These dynamic, adaptive, decentralised systems are used to process product lines with a shorter time to market, increased product diversity, greater specialisation, and shorter lifecycles. A number of government-backed initiatives have arisen in recent years to support this trend, including the German Industrie 4.0 initiative [4, 37], EU EFFRA Roadmap [2], US DMDII Strategic Plan [5], and the recent Japanese 'Connected Manufacturing' Industrial Value Chain Initiative [3]. Academic research in the area includes reconfigurable manufacturing systems [38, 22] with 'plug and produce' technologies [47, 9], holonic manufacturing [39, 41, 10], organic computing [44, 14], and the related evolvable production systems and evolvable assembly systems [46, 45, 47, 16, 48], among others.

The Evolvable Assembly Systems (EAS) project is investigating assembly systems that are self-adaptive [52] and able to support 'batch-size-of-one' production, where each product to be assembled may be unique [16]. In [20, 11], a formal approach was proposed to determine both *whether* a particular product can be assembled given a set of available assembly resources (the *realisability problem*), and *how* the product should be assembled using the resources (the *control problem*). The approach in that work takes as input the product in the form of a *recipe* specifying the operations necessary to assemble the product, and the *topology* or 'layout' of the production facility, which is generated offline from descriptions of the available assembly resources. Algorithms are given that convert a customer-requested product (recipe) into a *controller* at runtime, specifying the detailed steps to be executed by each assembly resource in the production facility. In [21] a tool implementing the controller synthesis algorithms in [20, 11] is briefly described. The tool outputs a controller in Business to Manufacturing Markup Language (B2MML) [6], a format suitable for execution by industrial assembly systems. In contrast to related work, the algorithms of [20, 11, 21] were motivated by the need to *rapidly* both check realisability and generate a controller for assembling the product.

The algorithms presented in [20, 11] for controller synthesis are abstract, and the work does not explain how they could be implemented in detail. In particular, [20, 11] (and also [21]) do not consider the key step of *generating* the topology from the descriptions of the resources comprising a production line, and simply assume the topology is available as an input to the synthesis algorithms. However, as topology generation is the most expensive step in the process [20], its scalability is critical to the feasibility of the whole approach. This is especially true for assembly systems that can transform, e.g. have resources or transport systems added or removed.

In this paper we give all algorithms (including topology generation) in sufficient detail to allow their reimplementation by other researchers, and report a preliminary experimental evaluation of the scalability of topology generation. As in [11], the data structures manipulated by the algorithms are specified in a textual encoding using the ISO-standard EBNF (Extended Backus-Naur Form) [1] notation. To evaluate the scalability of topology generation, we performed experiments to determine the time required to generate the topology as the number of assembly resources is increased in models of two practical production facilities: the SMC Pneumatics HAS-200 training platform described in [16], and the Precision Assembly Demonstrator (PAD) described in [11, 9]. Our results suggest that topology generation is feasible for small to medium sized transformable assembly systems.

This paper is organised as follows. We first outline the state of the art in Section 2, and then describe a simple assembly cell in Section 3 that serves as a running example throughout the remainder of the paper. Section 4 defines assembly resources and product recipes in EBNF notation. Section 5 describes the structure of an abstract controller and how it can be translated into a more concrete format for controlling industrial processes, and the methods for building a production topology, for checking realisability, and for synthesising an abstract controller. Section 6 evaluates the scalability of the algorithm for building the production topology. Finally, Section 7 summarises the paper and discusses future work.

2 State of the Art

The approach we propose is most closely related to that presented in [8], in that both approaches can check whether a product can be produced by a set of production resources. However there are important differences. Firstly, unlike [8], the work presented in this paper takes into account the parts that are being worked on by the production resources, and the correct routing of these parts between resources. Secondly, while the specification of the product and the controller (termed a "re-configuration" in [8]) required for producing it are both restricted to be a sequence of steps in the case of [8], the corresponding representations in this paper allow for the specification of parallel operations, tests and branching. Finally, the algorithms underlying the approach in [8] are only briefly mentioned in that paper, whereas

the algorithms that form the basis of our approach are the main focus of this paper.

There is also a strand of work on dynamically adding and removing production resources from a production line (i.e., 'plug and produce' systems), and on aggregating resource operations (or 'capabilities') to form composite operations at multiple levels of abstraction [9, 15, 12, 29, 34]. These are then matched against the desired product which is also specified as a sequence of composite operations. In contrast, our approach is not concerned with aggregation of operations: we assume that resource operations are specified at the same level of abstraction as the product specification, or that the transformation from high-level product specifications to low-level operations has already taken place. Moreover, in the body of work above, it is implicitly assumed either that production resources can service any order of operations in the product specification, e.g., parts can be transferred from any resource to any other resource (this is achieved via a shuttle conveyor system in [9] and a crane in [12]), or that separate software simulations will be run to check whether the order of operations and the routing of parts suggested by the generated controller are actually executable on the production system layout (as in [34]). In contrast, in our approach, the layout of the production system is modelled explicitly, and the order of operations and routing of parts specified by the controller is guaranteed to be executable by the production system.

Production systems have also been modelled as Petri Nets; see [54] for a survey. Both the product recipes and assembly resources considered in this paper could be modelled as Petri Nets, and the realisability of the recipe on the topology determined at runtime using Petri Net algorithms. However, such an approach is likely to be computationally intractable, even for restricted classes of Petri Nets, see, e.g., [26, 17, 35, 18]. In contrast, one of the key advantages of our approach is the ability to rapidly determine whether and how a product can be assembled (as we show in Section 6, this can be done in a few milliseconds). This is possible by virtue of a customised specification language for products and resources, which has lower expressive power than Petri Nets, but is sufficiently expressive for the assembly setting.

There is also work that uses customised languages for specifying processes. The most closely related approaches to ours include the Process Specification Language (PSL) [13], the plan specification languages of Hierarchical Task Network (HTN) planning systems [49], logic programming systems such as GOLOG [42], and Belief-Desire-Intention (BDI) agent systems [24]. The product specification language in this paper has some commonalites with such languages, including parameterised and concurrent processes, and specifying processes independently from the resources on which they are later executed. The main difference between our work and these languages, is that the latter were not designed to allow the rapid checking of whether a plan/process specification can be realised by the agent's resources. Indeed, as with Petri Nets, the computational overhead of the more expressive languages such as PSL are likely to be too great to support batch-size-of-one production.

From the remaining frameworks for production based on multiagent systems (see [40, 43] for comprehensive surveys on the topic), of particular relevance to our work are [19, 51, 32], where BDI-agent based frameworks are proposed that model the (manufacturing) environment as a dynamic set of 'artifacts', representing both physical entities such as resources on the shop floor, and computational entities such as web services. In [19], the product is specified 'declaratively' in terms of its machine-formable features, such as surface finish and colour, and any agent in the multiagent system can use semantic descriptions of currently available artifacts together with a declarative product specification to synthesise, on the fly, a production plan that is able to produce the product by requesting services from relevant artifacts. However, [19] generates production plans via first principles planning, which is intractable [30]. As a result, it is likely that the time required to generate a plan will rapidly become impractical as the number of artifacts increases. Similarly, plans in [32] are synthesised via HTN planning, which is more expressive but also more computationally expensive than first principles planning.

There are also other strands of work that focus on defining the product declaratively, such as [55, 56, 25, 28]. This specification is used by an algorithm that finds a combination of the processes offered by the available production resources that achieves the specification. The techniques used include planning a sequence of processes from first principles as above, and choosing from a set of predefined production rules, which associate processes to each feature in the specification. The work presented in this paper, on the other hand, assumes that the process combinations needed to build a product are devised and supplied by the operator in the form of a procedural product specification (product recipe).

Finally, we briefly consider the relationship to work on Joint Cognitive Systems (JCS) [33, 50]. Whereas traditional approaches to system design lead to "human-machine systems" in which humans supervise and control machines through interfaces, in a Joint Cognitive System humans and machines share cognitive functions in a collaborative manner. This can be applied in both product design [7] and system design and control [58, 36]. The relationship between JCS and the EAS project is briefly explored in [31]. In particular, the algorithms presented in this paper, along with the general approach from EAS, address a number of considerations raised in JCS system design [58, 31]:

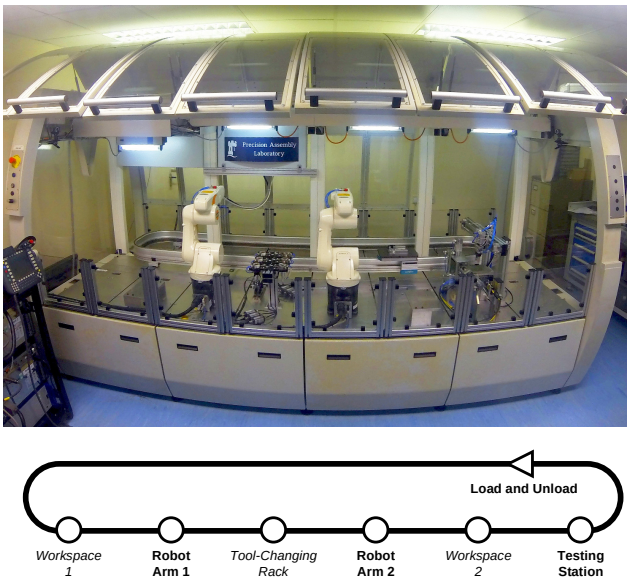

Figure 1: The Precision Assembly Demonstrator (top) and its layout (bottom).

decentralisation, self-organisation/adaptation, goal-orientation, capacity for evolution (in that goals can be updated over time), and enabling the system design process to take into account the relevant capabilities and resources.

3 An Experimental Assembly Cell

In this section, we briefly introduce the Precision Assembly Demonstrator (PAD) [9], which is used as a running example in the remainder of the paper. The PAD is based on the Modutec flexible assembly platform by Feintool Automation, and consists of six modules: a shuttle transport system linking modular stations to a loading/unloading area, two KUKA six-axis robotic arms with associated workspaces, a shared tool changing rack, and a testing and inspection station. The modules and their layout are shown in Figure 1.

The parts to be assembled into a product are mounted on a pallet. At the loading/unloading area, the system operator can place the pallet on a shuttle that runs along a linear transfer system. The pallet can be removed from the shuttle and placed at *Workspace 1*, where it can be worked on by *Robot Arm 1* before being returned to the shuttle. *Robot Arm 2* and *Workspace 2* are a mirror image of *Robot Arm 1* and *Workspace 1*. The *Tool-Changing Rack* is shared between the two robot arms and holds a number of end effectors, including grippers and suction tools, that are used for product assembly. The *Testing Station* allows mechanical and visual tests to be performed on the product once it has been assembled. The product is then returned to the loading/unloading area where an operator removes it.

In the example, the PAD is used to assemble a detent hinge for the cab interior of a truck. In the basic hinge, interior and exterior plastic leaves are fitted together and linked with a metal hinge pin. More complex hinges can be created by adding up to three metal ball–spring pairs within the interior leaf to adjust the engaging force. Additional end effectors may be used to apply glue to secure the hinge pin, or to engrave serial codes onto the leaves.

4 Assembly System Topologies and Product Recipes

In this section, we formally define the assembly resources that make up the assembly system, the topology which represents the layout of the resources, and product recipes which represent the products.

4.1 Assembly Resources and Assembly System Topologies

An *assembly resource* is a piece of hardware with a top-level controller (and potentially sub-controllers), which is typically a Programmable Logic Controller (PLC). The top-level controller controls the resource, sending signals to

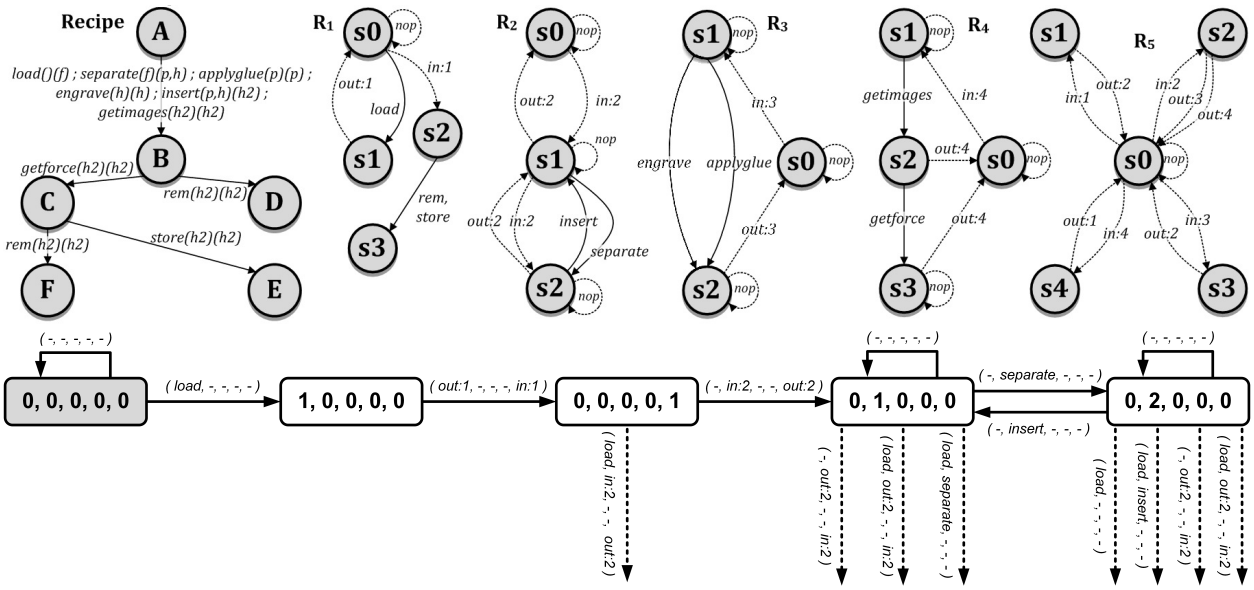

Figure 2: An example of a product recipe (top left), and simplified labelled transition systems $R_1, \ldots, R_5$ modelling the resources in our assembly platform (top right). States *s0* and *A* are initial states. A fragment of the associated production topology is shown in the bottom figure.

the hardware to perform actions, and collecting readings from sensors. Assembly resources are either production resources, which primarily perform *observable operations* on parts, or transport resources, such as a conveyor belt or shuttle system, which perform *internal operations* on parts, e.g. transporting parts between resources or logging data from sensors.

In [20, 11], assembly resources are defined as state diagrams, or more specifically, as labelled transition systems (LTSs), where nodes correspond to the states of the resource, and edges correspond to operations that can be performed in a state. For example, the LTSs representing the resources in the PAD are shown in Figure 2. The labels *in* and *out* represent *transfer operations*, which allow a part to be transferred between the resource performing the *out* operation and a resource performing the matching *in* operation. The *nop* labels denote special "no operation" tasks, that do nothing to parts and represent the resource idling. Resource R_1 is able to load a new pallet, transfer it to other resources for assembly, and then receive a (potentially partially-)completed product and either remove it for disposal or rework, or store it for delivery.[1] Resources R_2 and R_3 are the two robotic arms: R_2 can either take in a pallet on *in:2* and separate the hinge and pin, outputting each on a separate *out:2* transition, or accept each of those two parts on separate *in:2* transitions, insert the pin into the hinge barrel, and produce a single part on *out:2.* Resource R_3 can either engrave a serial number or apply glue to a given part, and R_4 is the testing station, which can perform a vision test and then (optionally) a force test. The vision test checks that the hinge is correctly assembled, so it must be performed before a force test, and it is not possible for the resource to idle between the two tests, hence the lack of a *nop* transition from state *s2*. R_5 is a transport resource that models the possible routing of parts by the shuttle system. The transitions represent the allowable routes between production resources; e.g., although a newly loaded pallet can be transported from R_1 to R_2, a freshly glued part cannot be transported from R_3 to R_4 in order to prevent it becoming stuck to the testing equipment. Observe that the *out* transfer operations of production resources (e.g. *out:1* in R_1) match with *in* transfer operations of transport resources (e.g. *in:1* in R_5) and vice-versa.

Together the resources form the *production topology* representing the layout of the assembly system. This is defined as the cross product of the LTSs representing the resources, excluding transitions with no matching 'in' and 'out' synchronisations (e.g., *out:1* in R_5 and *in:2* in R_2). A fragment of the topology obtained from the resources in Figure 2 is shown at the bottom of the figure. The vector of five numbers representing a topology state comprises the current state of each of the five resources, where "*(s0,s0,s0,s0,s0)*" (simplified to "*(0,0,0,0,0)*") is the initial topology state. Each topology transition is labelled with one vector of parallel resource operations that are possible from the corresponding resource states. For example, the topology transition labelled "*(-,separate,-,-,-)*" from state "*(0,1,0,0,0)*"

[1] Multiple transitions from one state to another are shown on a single edge, separated by commas.

indicates that one possible vector of parallel operations is where the second resource performs the 'separate' operation from state *s1* while all the other resources idle (dashes represent *nop* operations and commas represent parallelism). Note that the size of the topology, i.e., the number of topology transitions, grows exponentially as the number of resources increases [20]. Computing the topology thus takes time exponential in the number of resources. However, the topology is only computed once for a given assembly system layout. In Section 6 we present preliminary experimental results that demonstrate the scalability of our approach.

In this paper, we define resources, their operations and the production topology as EBNF [1] grammars (see Figure 3), as the EBNF representation corresponds more closely to the data structures used in the tool described in [21]. A resource is a couple consisting of the initial state of the resource and a set of resource transitions. A resource transition is a 3-tuple consisting of a possible state of the resource, an assembly operation that can be performed from that state, and the resulting resource state. An assembly operation is either the (name of) an observable operation, an internal operation, a transfer operation which transfers parts from one resource to another, or a special "no-op" operation indicating that the resource is idling. A topology is a couple consisting of the initial topology state, which constitutes the initial state of each resource, and a set of topology transitions representing the assembly and transport operations that can be performed in parallel by the individual assembly resources (from the initial and subsequent topology states). The **RESPARTS** element specifies the parts that are currently being held or worked on by each assembly resource.

One related representation for assembly resources in the context of cloud manufacturing is the one proposed in [27], which differs from the representation presented here in that the former explicitly represents the arguments of operations (parts and parameters) and allows distinguishing between operations that require a part to be loaded into the resource, from the same operation which can be performed on a part located in a shared working area. There is however no substantial difference, as one can easily encode those resources in the formalism presented here. Another related resource representation is the Function Blocks (FB) standard, discussed for instance in [57]. FBs are event-triggered components containing algorithms and an execution control chart, with inputs and outputs of data and events. Their nature is therefore different, as they are intended for modelling control software, treating the encapsulated behavior of a resource in a form that is similar to an electronic circuit. Similarly to our case, where an operation represents an arbitrarily complex process within a resource, the encapsulated behavior of an FB is treated as a black box, so that only the inputs and outputs are relevant. Our aim is analogous, but we abstract away from any specific technology, focusing instead on defining and solving the problem of orchestrating the resources (assessing realisability), to support automated reasoning. Indeed, approaches based on FBs rely on external production planning systems. Moreover, FBs are used to capture low level processes, whereas our goal is to describe processes at the same level of abstraction as the recipe. In our view, function blocks could constitute a viable implementation of the fine-grained execution control module of a concrete system, once realisability has been established and a controller generated.

4.2 Product Recipes

The products to be assembled by the assembly system are defined by product *recipes*. A recipe includes the constituent parts and the operations required to process and assemble these parts into the final product, any tests that must occur to verify the product during and at the end of the assembly process, and how to respond to the results of tests (e.g., whether a partially completed product should be reworked or discarded following a test). The recipes specify how, but not where, these operations and checks should be performed in order to assemble a given product.

In [20, 11], recipes are also formalised as LTSs, where nodes are states of parts in the assembly and edges represent composite operations as shown in Figure 2. A composite operation associated with a recipe transition represents the observable operations to be performed by the resources on their parts. Observable operations have parameters specifying the parts that are 'consumed' and 'produced' by the operation, and operations can be combined in sequence (denoted by the symbol ';') and/or in parallel (denoted by the symbol '||'). Each recipe transition is associated with a guard (test) that tests whether the part(s) meet certain properties (e.g., whether a hole has the correct depth), to determine whether the transition is applicable.[2] A guard is an observable operation specifying only input parts (which will be tested but not modified). While the guards on the outgoing transitions of a recipe state do not need to be mutually exclusive, they must cater for all the possible outcomes, e.g. where a hole has the correct depth and where it does not. This is straightforward to achieve as there can always be a 'default' guard, whose subsequent operation(s) in the transition notifies the human operator or discards the product, for example.

Figure 2 shows an example hinge recipe that is to be assembled by the PAD. The first composite operation (between states A and B) loads a new fixture pallet f, separates the pin p and hinge h, applies glue to p, engraves a serial number

[2]For transitions which do not require a test, a '*true*' guard is assumed.

⟨CompositeOp⟩ ::= ⟨Test⟩ (";" ⟨Step⟩)*

⟨Step⟩ ::= ⟨ParallelStep⟩ | ⟨ObservableOp⟩

⟨ParallelStep⟩ ::= ⟨ObservableOp⟩ ("||" ⟨ObservableOp⟩)+

⟨Test⟩ ::= ⟨ObservableOpName⟩ ⟨InputParts⟩

⟨ObservableOp⟩ ::= ⟨ObservableOpName⟩ ⟨InputParts⟩ ⟨OutputParts⟩ | ⟨ObservableOpName⟩ "()" ⟨OutputParts⟩

⟨AssemblyOp⟩ ::= ⟨ObservableOpName⟩ | ⟨InternalOp⟩

⟨InternalOp⟩ ::= ⟨InternalOpName⟩ | *nop* | ⟨Transfer⟩

⟨Transfer⟩ ::= *in* ":" *integer*+ | *out* ":" *integer*+

⟨InputParts⟩ ::= "(" ⟨Parts⟩ ")"

⟨OutputParts⟩ ::= "(" ⟨Parts⟩ ")"

⟨Parts⟩ ::= ⟨Part⟩ ("," ⟨Part⟩)*

⟨Part⟩ ::= *string*

⟨ObservableOpName⟩ ::= *unique string*

⟨InternalOpName⟩ ::= *unique string*

⟨Recipe⟩ ::= ⟨InitRecipeState⟩ "{" ⟨RecipeTrans⟩ ("," ⟨RecipeTrans⟩)* "}"

⟨RecipeTrans⟩ ::= "(" ⟨RecipeState⟩ "," ⟨CompositeOp⟩ "," ⟨RecipeStateTo⟩ ")"

⟨InitRecipeState⟩ ::= ⟨RecipeState⟩

⟨RecipeState⟩ ::= *string*

⟨Resource⟩ ::= ⟨InitResourceState⟩ "{" ⟨ResourceTrans⟩ ("," ⟨ResourceTrans⟩)* "}"

⟨ResourceTrans⟩ ::= "(" ⟨ResourceState⟩ "," ⟨AssemblyOp⟩ "," ⟨ResourceStateTo⟩ ")"

⟨InitResourceState⟩ ::= ⟨ResourceState⟩

⟨ResourceState⟩ ::= *string*

⟨Topology⟩ ::= ⟨InitTopState⟩ "{" ⟨TopTrans⟩ ("," ⟨TopTrans⟩)* "}"

⟨TopTrans⟩ ::= "(" ⟨ResStates⟩ "," ⟨ResOps⟩ "," ⟨ResStatesTo⟩ ")"

⟨InitTopState⟩ ::= ⟨ResStates⟩

⟨ResStatesTo⟩ ::= ⟨ResStates⟩

⟨ResStates⟩ ::= ⟨ResourceState⟩+

⟨ResOps⟩ ::= ⟨AssemblyOp⟩ ("||" ⟨AssemblyOp⟩)*

⟨ResParts⟩ ::= ⟨ResourceParts⟩+

⟨ResourceParts⟩ ::= "(" ⟨Parts⟩ ")" | "()"

Figure 3: The EBNF syntax of operations (left), and of recipes, resources and topologies (right).

on h, and inserts p into h to form $h2$, which is then visually examined, generating data. At this point, a test is performed against the vision data (the guards are not shown in order to improve the readability of the diagram), and based on the result the product is either removed from the system or a force analysis is performed. The outcome of performing a test against the force data generated similarly determines whether the product is removed or sent to storage for delivery.

In this paper we define recipes and their associated (composite) operations as EBNF grammars (see Figure 3). Composite operations represent the observable operations to be performed by the resources on the parts, and have parameters specifying the parts that are consumed and produced by the operation. A recipe is a set of recipe transitions, where each recipe transition is a 3-tuple consisting of a recipe state, a composite operation and the resulting state of the recipe.

5 Realisability Assessment and Controller Generation Methodology

In this section, we describe how a production topology is computed, and how the *realisability* of a product recipe (i.e., whether the product can be assembled) on a precomputed topology is determined. We start with an informal definition of what it means for a product recipe to be realisable on a production topology, or in other words, *when* it is possible to assemble a product using a particular assembly system.

A recipe is realisable on a topology if it is possible to associate states in the recipe with states in the topology, such that each transition (composite operation) in the recipe can be executed by transitions (assembly operations) in the topology (taking into account the parts currently present in the resources), and the same is possible for the entire recipe, irrespective of the outcome of guards and the 'path' through recipe transitions that might be followed at runtime. If a recipe is realisable on a topology, then there is an associated controller which specifies how resources may be orchestrated in order to assemble the product. Intuitively, a *controller* is a state diagram with the same structure as the recipe. A controller is obtained by annotating each recipe state with the associated topology state (i.e., resource states) and the parts being worked on in that state, and annotating each recipe transition with the sequence of topology (i.e., resource) transitions that need to be carried out in order to execute the recipe transition.

As an example, Figure 4 shows a controller for the recipe and topology in Figure 2. Each controller transition is labelled with the composite operation in the corresponding recipe transition (in bold), and the sequence of topology transitions (shown in italics from top to bottom) that are needed to execute the composite operation. In the interests of

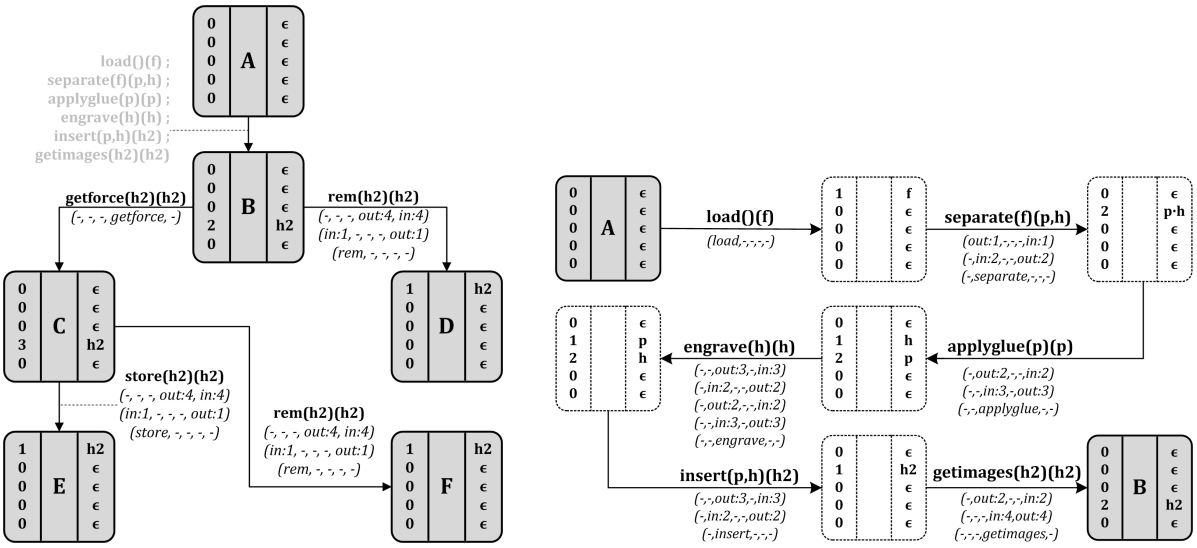

Figure 4: A controller (left) for the recipe and topology in Figure 2, and the controller's first transition (right).

readability, the transitions necessary to execute the first composite operation, between the first two recipe states A and B, are shown on the right of the figure: the white states represent "intermediate" states between recipe states A and B, and transitions depict the topology transitions needed to execute the individual observable operations comprising the first composite operation in the controller. Each parenthesised row below a transition represents a transition of the topology, and thus each element within such parentheses represents an operation to be performed in parallel by the corresponding resource. The five vertical numbers in the left column of each controller state represent the states of the five resources comprising the topology in that controller state; the strings in the right column correspond to the parts held by each resource (the symbol ϵ indicates that no parts are held by the corresponding resource).

5.1 Controller Generation for Industrial Assembly Processes

Controllers are expressed in Business to Manufacturing Markup Language (B2MML) [6]. B2MML implements the ANSI/ISA-95 (Enterprise-Control System Integration) standards in XML, and is increasingly being adopted in manufacturing [23]. B2MML defines the data and process models that form the interface to manufacturing execution systems (MESs) [53], which manage and monitor work-in-progress on low-level shop floor control systems, for example, using PLCs. B2MML thus provides higher-level enterprise control systems with a convenient interface to hardware.

The part of B2MML relevant to this paper is the *operations-schedule*. The operations-schedule specifies each operation to be performed via an *operations-request* that captures the Bill of Process, Bill of Materials, and Bill of Tooling within a 'segment requirement'. The operations-request specifies the parameters to be passed to the equipment controllers (such as the specific gripper or drill bit to use, or the gripper separation or depth of cut required), the materials requirements (i.e., the parts required and produced by the operation), and the equipment requirements (i.e., the required assembly resources for each step). An operations-schedule combines operations-requests into an AND-OR tree which represents an "unfolding" of the state diagram that encodes the controller. More precisely, whenever multiple transitions in the state diagram have the same end-node, the node is duplicated for each transition. This translation from a state diagram into a tree is necessary because the XML format used by B2MML encodes data in a hierarchical structure, starting from a root node. In addition to this transformation, creating an operations-schedule requires extracting certain details from the controller, such as: *(i)* the equipment requirements, *(ii)* the materials requirements, and *(iii) when* parts need to be transferred and *which* pairs of resources need to work together in order to achieve this. Such parallel 'transfer' operations within a controller transition are represented as a single 'abstract' operations-request within the operations-schedule, as B2MML does not support explicit parallelism. During execution, an abstract operations-request is treated by the PLC as two separate parallel operations-requests when commands are sent to hardware controllers.

The execution semantics of an operations-schedule is as follows. On reaching a choice point during execution, the

⟨OperationsRequest⟩
⟨OperationsType⟩ **Loading** ⟨/OperationsType⟩
⟨ID⟩ **Load** ⟨/ID⟩
⟨Desc⟩ **Load a fresh part** ⟨/Desc⟩
⟨SegmentReqmnt⟩
⟨ID⟩ **load()(f)** ⟨/ID⟩
⟨Desc⟩ **First instance of the load type** ⟨/Desc⟩
⟨SegmentParam⟩
⟨ID⟩ **Integer** ⟨/ID⟩
⟨Value⟩ **3** ⟨/Value⟩
⟨Desc⟩ **Load from third shelf** ⟨/Desc⟩
⟨/SegmentParam⟩
⟨MaterialReqmnt⟩
⟨MaterialUse⟩ **Produced** ⟨/MaterialUse⟩
⟨Quantity⟩ **1** ⟨/Quantity⟩
⟨MaterialReqmntProperty⟩
⟨ID⟩ **f** ⟨/ID⟩
⟨Value⟩ **Fixture** ⟨/Value⟩
⟨/MaterialReqmntProperty⟩
⟨/MaterialReqmnt⟩
⟨/SegmentReqmnt⟩
⟨/OperationsRequest⟩

⟨OperationsRequest⟩
⟨OperationsType⟩ **Production** ⟨/OperationsType⟩
⟨ID⟩ **Separate** ⟨/ID⟩
⟨Desc⟩ **Separate pin and hinge from pallet-fixture** ⟨/Desc⟩
⟨SegmentReqmnt⟩
⟨ID⟩ **separate(f)(p,h)** ⟨/ID⟩
⟨Desc⟩ **First instance of the separate type** ⟨/Desc⟩
⟨SegmentParam⟩
⟨ID⟩ **String** ⟨/ID⟩
⟨Value⟩ **Gripper3** ⟨/Value⟩
⟨Desc⟩ **Use Gripper3 from tool rack** ⟨/Desc⟩
⟨/SegmentParam⟩
⟨MaterialReqmnt⟩
⟨MaterialUse⟩ **Consumed** ⟨/MaterialUse⟩
⟨Quantity⟩ **1** ⟨/Quantity⟩
⟨MaterialReqmntProperty⟩
⟨ID⟩ **f** ⟨/ID⟩
⟨Value⟩ **Fixture** ⟨/Value⟩
⟨/MaterialReqmntProperty⟩
⟨/MaterialReqmnt⟩
⟨MaterialReqmnt⟩
⟨MaterialUse⟩ **Produced** ⟨/MaterialUse⟩
⟨Quantity⟩ **1** ⟨/Quantity⟩
⟨MaterialReqmntProperty⟩
⟨ID⟩ **p** ⟨/ID⟩
⟨Value⟩ **Pin** ⟨/Value⟩
⟨/MaterialReqmntProperty⟩
⟨/MaterialReqmnt⟩
⟨MaterialReqmnt⟩
⟨MaterialUse⟩ **Produced** ⟨/MaterialUse⟩
⟨Quantity⟩ **1** ⟨/Quantity⟩
⟨MaterialReqmntProperty⟩
⟨ID⟩ **h** ⟨/ID⟩
⟨Value⟩ **Hinge** ⟨/Value⟩
⟨/MaterialReqmntProperty⟩
⟨/MaterialReqmnt⟩
⟨/SegmentReqmnt⟩
⟨/OperationsRequest⟩

⟨OperationsSchedule⟩
⟨PublishedDate⟩ 2016-11-24;12:38:50 ⟨/PublishedDate⟩
⟨OperationsRequest⟩
⟨OperationsType⟩ **Loading** ⟨/OperationsType⟩
⟨ID⟩ **Load** ⟨/ID⟩
⟨Desc⟩ **Load a fresh part** ⟨/Desc⟩
⟨SegmentReqmnt⟩
⟨ID⟩ **load()(f)** ⟨/ID⟩
⟨Desc⟩ **First instance of the load type** ⟨/Desc⟩
⟨SegmentParam⟩ ··· ⟨/SegmentParam⟩
⟨MaterialReqmnt⟩ ··· ⟨/MaterialReqmnt⟩
⟨EquipmentReqmnt⟩
⟨EquipmentID⟩ **Resource1** ⟨/EquipmentID⟩
⟨EquipmentReqmntProperty⟩ ⟨ID⟩ **Loader** ⟨/ID⟩
⟨/EquipmentReqmntProperty⟩
⟨/EquipmentReqmnt⟩ ···
⟨/SegmentReqmnt⟩
⟨/OperationsRequest⟩
⟨OperationsRequest⟩
⟨OperationsType⟩ **Routing** ⟨/OperationsType⟩
⟨ID⟩ **TransferType** ⟨/ID⟩
⟨Desc⟩ **Parallel transfer from loader to conveyor** ⟨/Desc⟩
⟨SegmentReqmnt⟩
⟨ID⟩ **TransR1toR5** ⟨/ID⟩
⟨Desc⟩ **First instance of transfer** ⟨/Desc⟩
⟨SegmentParam⟩ ··· ⟨/SegmentParam⟩
⟨MaterialReqmnt⟩ ··· ⟨/MaterialReqmnt⟩
⟨EquipmentReqmnt⟩
⟨EquipmentID⟩ **Resource1** ⟨/EquipmentID⟩
⟨EquipmentReqmntProperty⟩ ⟨ID⟩ **Loader** ⟨/ID⟩
⟨/EquipmentReqmntProperty⟩
⟨/EquipmentReqmnt⟩
⟨EquipmentReqmnt⟩
⟨EquipmentID⟩ **Resource5** ⟨/EquipmentID⟩
⟨EquipmentReqmntProperty⟩ ⟨ID⟩ **Conveyor** ⟨/ID⟩
⟨/EquipmentReqmntProperty⟩
⟨/EquipmentReqmnt⟩
⟨/SegmentReqmnt⟩
⟨/OperationsRequest⟩
⟨OperationsRequest⟩
⟨OperationsType⟩ **Routing** ⟨/OperationsType⟩
⟨ID⟩ **TransferType** ⟨/ID⟩ ···
⟨SegmentReqmnt⟩
⟨ID⟩ **TransR5toR2** ⟨/ID⟩ ···
⟨/SegmentReqmnt⟩
⟨/OperationsRequest⟩
⟨OperationsRequest⟩
⟨OperationsType⟩ **Production** ⟨/OperationsType⟩
⟨ID⟩ **Separate** ⟨/ID⟩ ···
⟨/OperationsRequest⟩ ···
⟨ChoicePoint⟩
⟨ChoicePoint⟩ ··· ⟨/ChoicePoint⟩
⟨Sequence⟩ ··· ⟨/Sequence⟩
⟨/ChoicePoint⟩
⟨/OperationsSchedule⟩

Figure 5: A B2MML operations-request for the $load()(f)$ operation in the recipe in Figure 2 (top left); a B2MML operations-request for the $separate(f)(p,h)$ operation in the recipe in Figure 2 (bottom left); and a B2MML operations-schedule combining the two operations-request specifications. Some details are ommitted for brevity.

first operations-request on each outgoing branch (which corresponds to a guard (test) operation) is executed until one of the test operations returns 'success'. Then, the rest of the operation's branch is executed until the next choice point is reached. This process continues until a complete 'path' through the operations-schedule has been executed.

An example operations-schedule is shown in Figure 5, which shows how the two operations-requests on the left are connected by a transfer operation (the *TransR1toR5* operations-request). Note that the same assembly resources that were allocated to the operations in the controller in Figure 4 appear in the operations-schedule as equipment

requirements (within ⟨EQUIPMENTID⟩ tags), and the parts that are consumed and produced by the operations appear as materials requirements (within ⟨MATERIALREQMNT⟩ tags). The first (outer) ⟨CHOICEPOINT⟩ tag toward the bottom of the operations-schedule corresponds to the first 'choice point' (state B) in the controller, and the nested ⟨CHOICEPOINT⟩ tag in the operations-schedule corresponds to the second 'choice point' (state C) in the controller. The ⟨SEQUENCE⟩ tag in the operations-schedule corresponds to the sequence of topology transitions that are associated with the controller's transition from state B to D.

5.2 Decision Making Methods

We now present flowcharts describing how a production topology is computed, and how the realisability of a product recipe on a topology is determined. The corresponding algorithms can be found in the appendix. The flowcharts also illustrate how sequences of topology transitions corresponding to individual composite operations in a recipe are stored as part of a controller that specifies how the available assembly resources should be orchestrated in order to assemble the product. The flowcharts (and the corresponding algorithms in the appendix) use the EBNF data structures defined in Figure 3, and are based on the abstract algorithms in [20].

To compute the topology, we start from an empty set of topology transitions and then incrementally add (only) valid transitions to the set. More precisely, the function ComputeTop shown in the flowchart in Figure 6 is invoked with the following two inputs: the empty set of topology transitions, and the topology state **INITTOPSTATE**, comprising the initial state of each resource. The flowchart picks an outgoing resource transition for each state in **RESSTATES**, and forms a single topology transition **TOPTRANS**, which includes the resulting resource states within a single (resulting) topology state. As in [20, 21], a topology transition formed from the combination of resource transitions is valid only if the latter have either matching *in* and *out* transfer operations, or no transfer operations. For example, consider the resource R_1 in Figure 2. Transition *in:1* from state *s0* can only be performed in parallel with the matching transition *out:1* when resource R_5 is in state *s4*. Since, in the initial state of the topology, all resources start in their initial state (*s0*), such a joint transition from the initial state cannot form part of the topology. The same process is carried out recursively from the resulting topology state by calling ComputeTop with parameter **RESSTATESTO**; after this the remaining vectors of resource transitions associated with **RESSTATES** are considered ('loop'). In order to ensure that the recursive calls to the flowchart eventually terminate, all the vectors of resource states that have been considered are marked as 'visited',[3] and then checked at each recursive call to avoid reconsidering them.

Generating the topology in this way avoids computing all possible combinations of resource states and all possible combinations of resource operations. Moreover, it ensures that only states reachable from the initial state of the topology are generated. In Section 6 we evaluate the performance of the topology generation flowchart as the number of assembly resources is increased for models of two practical production facilities.

Next, we provide flowcharts describing how the realisability of a product recipe **RECIPE** is determined on a production topology **TOPOLOGY** computed as above. The top-level flowchart, Figure 7 works as follows. Given a state of the recipe **RECIPESTATE** (e.g., the initial state of the recipe) and a vector of resource states **RESSTATES** (i.e., a state of the topology) representing the state of the entire production facility at a given point in time, the flowchart determines, for each recipe transition of the form

$$(\textbf{RECIPESTATE}, \textbf{COMPOSITEOP}, \textbf{RECIPESTATETO}),$$

whether the available resources can execute **COMPOSITEOP** from their corresponding states as indicated in **RESSTATES**. To this end, the flowchart in Figure 8 iterates over topology transitions from **RESSTATES**, i.e., the vectors of (parallel) assembly operations **RESOPS** that are executable from **RESSTATES** by the various resources, to find a vector that *can* perform the first (non-guard) step in **COMPOSITEOP**. More precisely, each topology transition of the following form is considered:

$$(\textbf{RESSTATES}, \textbf{RESOPS}, \textbf{RESSTATESTO}),$$

where the last element is the vector of resource states that result from executing **RESOPS**. Checking whether the first step (**FIRSTOP**) in **COMPOSITEOP** matches **RESOPS** amounts to checking whether the former can be *allocated* to **RESOPS** (using function Allocate). This holds if two conditions are met: *(i)* there is a one-to-one mapping from the names of the parallel recipe operations to the names of the parallel resource operations, and *(ii)* the parts required as input to the recipe operations are consistent with **RESPARTS**, i.e., currently present in the resources that map to those operations. If the allocation is successful, then **RESPARTS2** (is not *false* and) contains the vector of parts that will be

[3] This trivial process of marking has been omitted from the flowchart for readability.

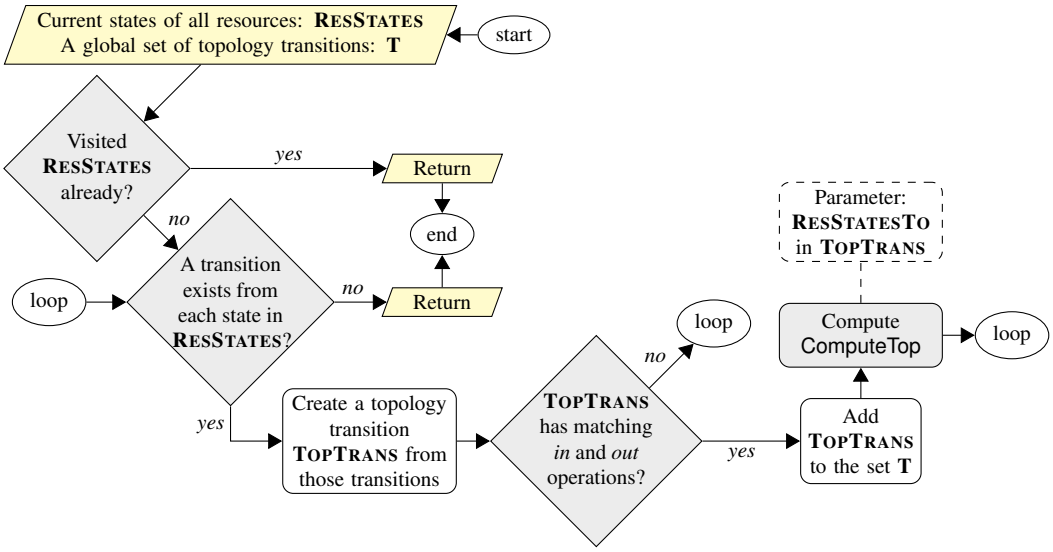

Figure 6: The flowchart for function ComputeTop. A dashed rectangle is a comment, a normal rectangle depicts a processing step, a coloured rectangle depicts the invocation of a function defined as a flowchart, a diamond depicts a decision point, a parallelogram depicts an input/output, and an ellipse depicts the start/end or a section of a flowchart.

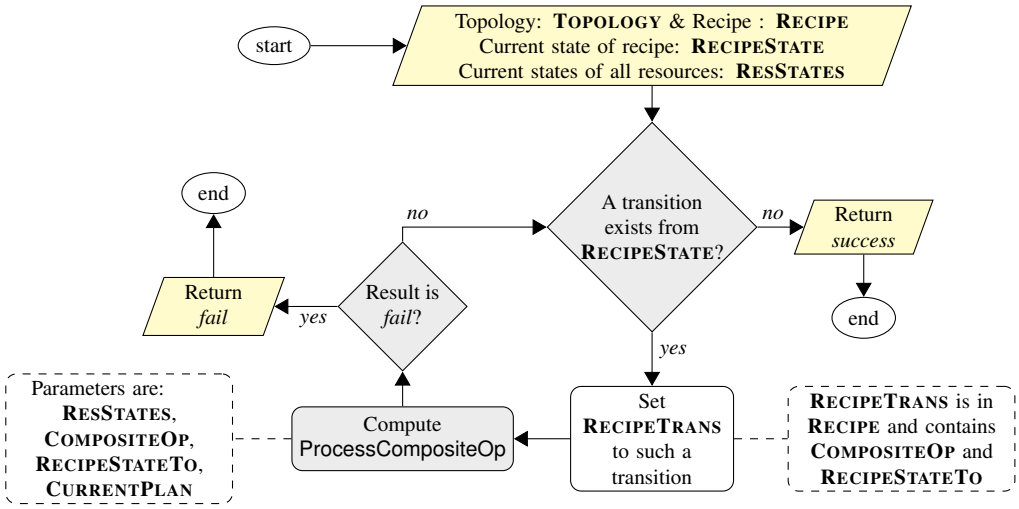

Figure 7: The flowchart for function CheckManuf. This is called in the flowchart of Figure 8, and vice versa.

produced by the recipe operations (e.g., the input parts with some holes in them). If **RESOPS** indicates that parts need to be transferred between pairs of resources, i.e., it has pairs of matching *in* and *out* transfer operations (e.g. *in:3* and *out:3*), one part per pair is transferred (using function TransferParts) from the resource performing the *out* operation to the one performing the *in*; the new locations of the parts in the assembly system are then referred to as **RESPARTS2**.

Next, the vector of resource states **RESSTATESTO**, i.e., the states that would result from performing the vector of resource operations **RESOPS**, is annotated with the new locations of parts (using function AnnotateParts), and **RESOPS** is appended (using function Append) to the current plan **CURRPLAN** that is being built (which is later stored in the controller if the composite operation being considered does turn out to be executable, i.e., it is processed completely). The flowchart is then followed recursively in order to check the executability of the remaining steps in **COMPOSITEOP** above, until there are no steps left to process.

In the case where the allocation of **FIRSTOP** to **RESOPS** fails because the latter only contains internal operations, such as maintenance or transfer operations (to which it will not be possible to allocate the *observable* parallel operations in **FIRSTOP**), the flowchart checks whether the first and subsequent steps in **COMPOSITEOP** could instead be

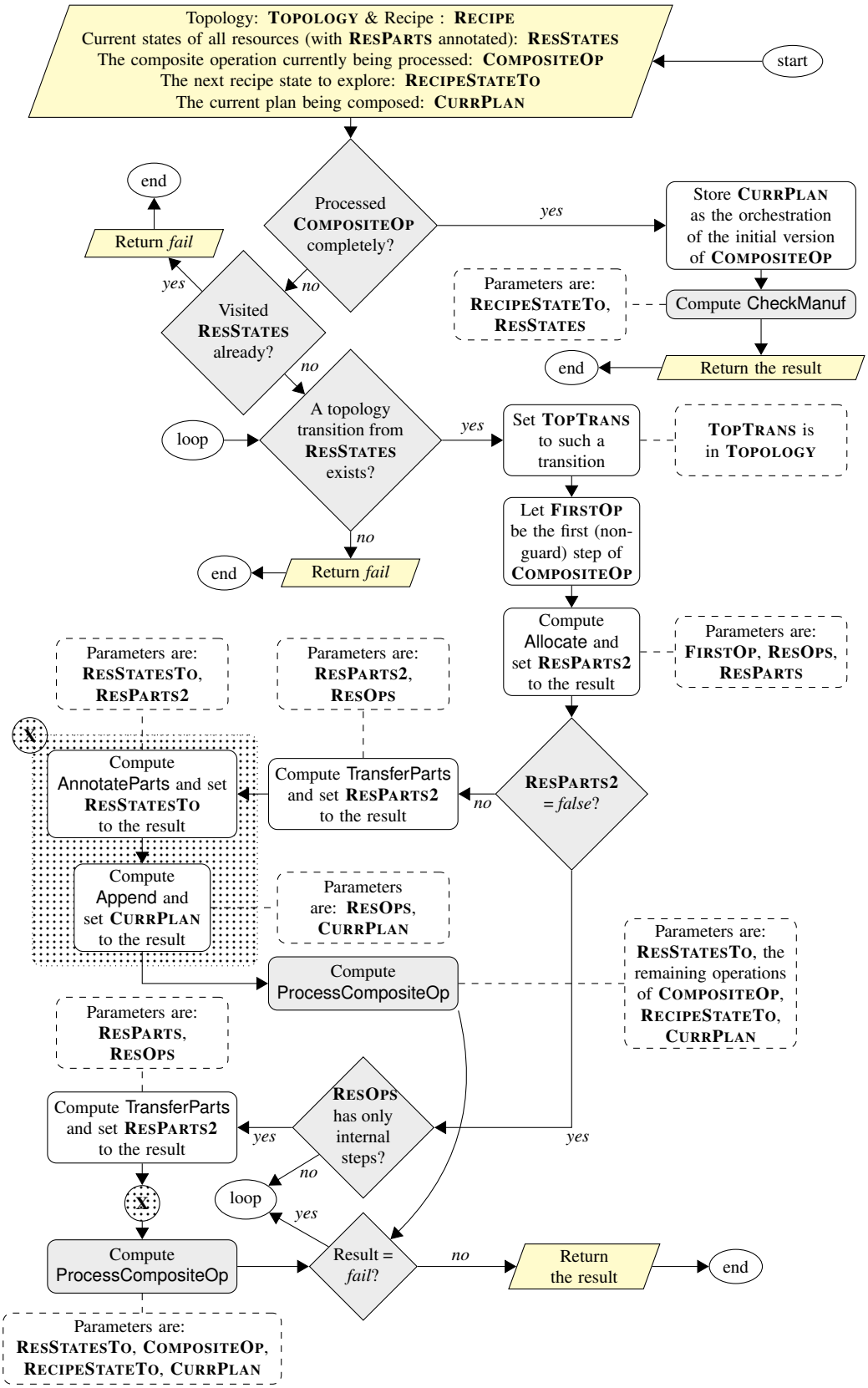

Figure 8: The flowchart for function ProcessCompositeOp. The circled **X** refers to the steps in the dotted rectangle.

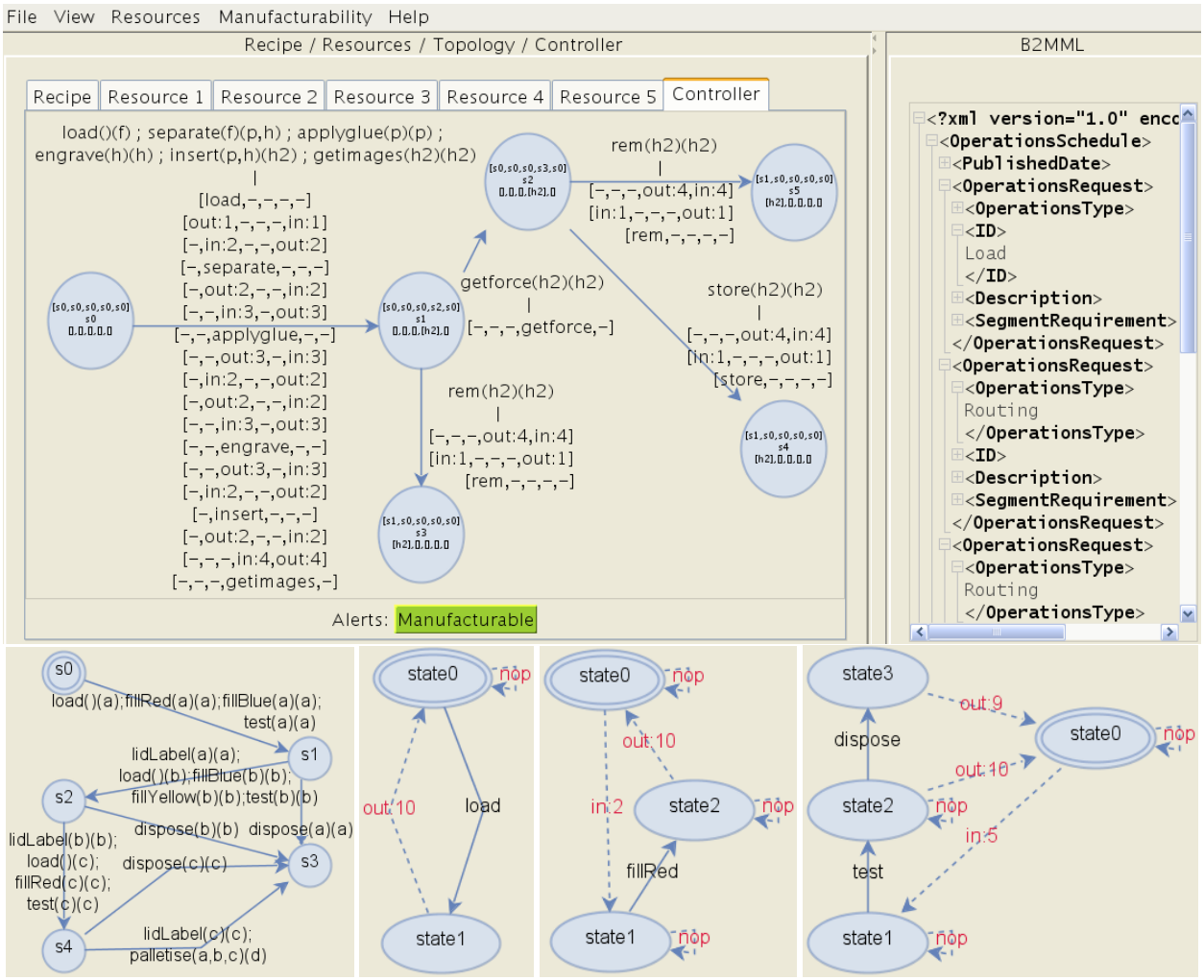

Figure 9: A controller (solution) for producing the hinge recipe on the PAD (top), and a product recipe, loader, filler, and tester corresponding to the SMC platform (bottom), respectively.

allocated once the internal operations—including part transfers if any—have completed, i.e., to a vector of parallel resource operations that are possible from **ResStatesTo** (relative to **ResParts2**).

6 Implementation and Experimental Results

We have implemented the flowcharts described above as part of a software tool that can be used by an operator of a production facility to model assembly resources, and to compute and visualise the production topology, a controller, and its corresponding B2MML operations-schedule. Screenshots of LTSs modelled in the tool are shown in Figure 9. The figure also shows a controller for the hinge recipe and its corresponding operations-schedule, which correspond to the controller in Figure 4 and the operations-schedule in Figure 5. In this section, we use the tool to empirically evaluate the feasibility of our approach.

The computational complexity of checking the realisability of a recipe and synthesising a controller is known to be polynomial on the size of the topology and exponential in the size of the recipe and number of resources [20], and the time required for the tool to check realisability and generate a controller (reported in [21]) is on the order of milliseconds. For example, given a precomputed topology for a 5-resource version of the PAD, the time required to generate a controller for the hinge recipe is 2 milliseconds [21]. Compared to the time required to actually assemble the hinge, this is sufficiently small as to allow batch size of one production, given a precomputed topology. However, the

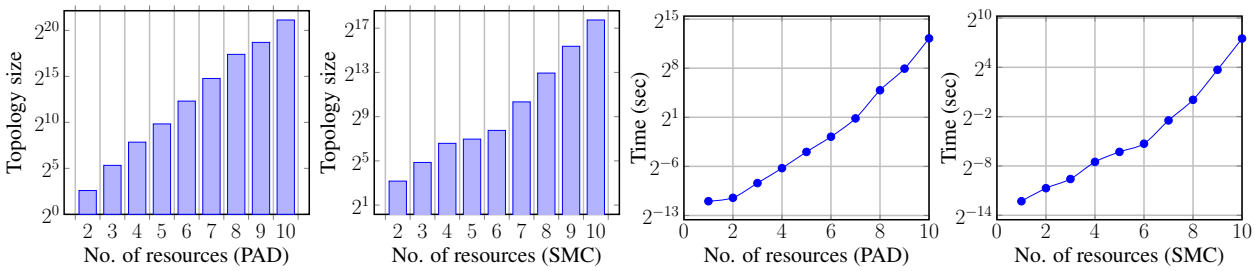

Figure 10: The size of the computed topology (log-scale) with an increasing number of resources for the PAD and SMC (the two graphs on the left), and average CPU time (log-scale) with an increasing number of resources for the PAD and SMC (the two graphs on the right).

time required to compute the topology has not previously been reported. The computational complexity of generating the topology from descriptions of the assembly resources is known to be exponential in the number of resources, and polynomial in their size [20]. It is therefore possible that topology generation may be prohibitively expensive, even for production facilities consisting of a small number of assembly resources, or where the set of resources or the allowable routes for the transfer of parts between them changes frequently.

To evaluate the scalability of topology generation, we performed experiments to determine the time required to generate the topology as the number of assembly resources is increased for models of two practical assembly systems: the PAD described in [9, 11] and summarised in Section 3, and the SMC Pneumatics HAS-200 training platform described in [16]. The SMC platform has 8 stations connected by a ring-shaped conveyor belt, which are configured as follows. The first station loads containers onto the conveyor belt as orders come into the production line. The next three stations are blue, red and yellow particulate dispensers, respectively. These stations pick up the container, add the necessary quantity of particulate (by weight), and return the container to the conveyor belt. The next two stations are testing stations which perform the same task of measuring the volume of particulate in the container, and will reject containers into a waste bin if they fail the test. The next station fits a lid to the container, prints a custom label, and attaches it to the top of the container. The last station is a palletisation station which removes completed containers from the system for packaging. All stations are able to let a container simply pass through to the next station along the belt, without performing any operations on the container. A product recipe can request containers with different quantities of blue, red, and yellow coloured particulate, where each container could represent a meal, pill, or blister pack, and the particulate is representative of ingredients or individual pills. Figure 9 shows the main LTSs for the SMC platform and an example product recipe that is realisable on the corresponding topology.

For the PAD experiments, we increased the number of resources beyond the 5 original ones by duplicating some of the pre-existing production resources in a useful way and connecting them to the transport resource. In particular, the 6-resource version of the PAD had a duplicate of R_3, i.e., the gluing/engraving robot, which enabled doing these operations in parallel on the hinge and pin. The larger versions of the PAD had a duplicate of either resource R_1, R_2, or R_4, in the PAD version one size smaller, and the largest version of the PAD comprised 10 resources. Since the SMC already had 9 resources (including the conveyor belt), only one resource was added to obtain the largest version of the SMC. The new resource modelled a station for collecting and storing containers that were rejected by the testing stations. The experiments were performed on a Lenovo ThinkPad with a quad-core Intel Core i7 processor and 20GB of RAM, running Windows 7 Enterprise SP1 64-bit and Java 1.8.

The two graphs on the left in Figure 10 show the size of the topology (the number of topology transitions) for the PAD and SMC as the number of resources increases.[4] For example, the topology for the PAD with 10 resources contains about 2.3 million transitions.

The two graphs on the right in Figure 10 show the CPU time required to compute the topology for the PAD and SMC as the number of resources increases. Each data point represents an average over 10 runs.[5] As can be seen, the time required to compute the topology increases with the number of resources, and the increase is greater for the PAD than the SMC. Computing the topology for the 10-resource PAD required about 2^{12} seconds (81 minutes) on average. Given that the topology only needs to be computed when the set of assembly resources changes, e.g., when a new resource is added, these times suggest that recomputing the topology in such circumstances is feasible, at least for

[4]Data is for a single run, as the topology size does not change for a given input.

[5]The standard deviations are omitted from the graphs as they are too small to be noticeable.

small to medium sized production facilities.

While these results are encouraging, it is important to note that care is required in modelling the assembly resources to minimise the time required to compute the topology and its size. For example, in the SMC model shown in Figure 9, there was no need for a *nop* transition from *state1* of the loading station, as a loaded part can be immediately sent out to some other resource via transfer operation *out:10* (and temporarily held in the latter resource until an operation needs to be performed on the part). Adding just the single *nop* transition—which loops back to *state1*—results in a topology that is 24% larger in the 10-resource model of the SMC.

7 Conclusion and Future Work

In this paper we have developed and implemented methods for computing a production topology, assessing realisability, and generating a controller, based on the formal approach presented in [20, 11]. Our approach takes as input a *recipe* model specifying the operations necessary to assemble a product and the *topology* or 'layout' of the production facility generated from models of the *assembly resources* comprising the facility, and transforms the recipe into a *controller* specifying the detailed steps to be executed by each assembly resource. The controller is then translated into B2MML [6], a format suitable for execution by industrial assembly systems. Crucially, we give the methods for topology computation, realisability assessment, and controller generation in sufficient detail to allow their reimplementation by other researchers. The methods are presented in the form of flowcharts, and the data structures are specified in ISO/IEC EBNF [1] notation.

In addition, we report a preliminary experimental evaluation of the scalability of topology generation. Topology generation is the most expensive step in the process [20], and its scalability is critical to the feasibility of the whole approach. To evaluate the scalability, we performed experiments to determine the time required to generate the topology as the number of assembly resources are increased for models of two practical production facilities: the SMC Pneumatics HAS-200 training platform described in [16], and the Precision Assembly Demonstrator described in [9, 11]. Our results suggest that topology generation is feasible for small to medium sized production facilities.

The core modules of the tool described in this paper are used for orchestrating the activities of agents in the Evolvable Assembly Systems (EAS) architecture, an agent-based architecture for assembly control software designed to address rapidly changing product and process requirements [16]. In the EAS architecture, a product recipe that is received is handled by an individual software agent, that first checks that the recipe is realisable and then synthesises a controller for orchestrating the agents.

While we have shown that our algorithms are feasible for small to medium sized production facilities, they are unlikely to be useful in large production facilities whose layouts change frequently. This is due to the computational complexity of recomputing the production topology. One line of future work could be to identify situations in which the topology can be *adapted* (rather than recomputed) when the assembly system layout changes. For example, if a resource at one end of an assembly line is removed, it is sometimes possible to simply delete the relevant topology transitions (e.g. those corresponding to operations that transfer a part to the removed resource and then work on the part) from the production topology, and then remove the nop operation corresponding to the resource in each remaining topology transition.

A second limitation is that, while we are able to check whether a product recipe is realisable on the production topology, and, if it is, synthesise a controller, we do not address the problem of what to do if the recipe is *not* realisable on the topology. One solution could be to develop an algorithm that synthesises an 'approximate controller', which assumes that a recipe operation that is missing in a resource (or a number of such operations, up to a predefined maximum) could be installed by the engineer later as a new process. To facilitate this, the controller could indicate the operation that was missing and the state in the resource from where the operation was assumed to be possible. For example, if the recipe in Figure 9 and the corresponding controller have the $fillGreen(a)(a)$ operation instead of $fillRed(a)(a)$, the engineer might be able to quickly create the associated process in the second SMC station, and plug in a green particulate dispenser (assuming that the controller relied on the second station to have the process). Alternatively, if all the operations specified in the recipe are provided by the resources comprising the topology, but the recipe is not realisable, we could check if an alternative layout of the resources (i.e., reconfiguring the transport resources) would allow the product to be assembled, by checking for the existence of a controller if the transport resources form a complete graph. If this renders the recipe realisable, the transfer operations used by the controller in the "relaxed topology" indicate how the layout of the resources could be reconfigured to allow the product to be assembled. A similar approach could be used to determine if, for example, a recipe could be assembled in fewer steps

or at lower cost by changing the layout, that is, whether the current topology is the most appropriate for a particular product. Clearly, such reconfiguration of the plant would only be worthwhile for larger batch sizes and/or where the cost of assembly is high.

Finally, there are also some other interesting directions for future work. First, since this work checks realisability against an 'empty' production line, i.e., where no parts or partially assembled products are initially present, future work could take into account the products that are currently being assembled, and generate a controller that can essentially 'interleave' the new product recipe with the other products being assembled. Second, a set of algorithms could be developed that synthesises multiple controllers (or all the possible ones) for assembling a product, and then chooses the most desirable one, e.g. the one that minimises the makespan.

Acknowledgements

The authors are grateful for support by the Engineering and Physical Sciences Research Council via grants EP/K018205/1 and EP/K014161/1.

References

[1] Information technology - syntactic metalanguage - Extended BNF. `http://standards.iso.org/ittf/PubliclyAvailableStandards/`, 1996 (accessed March 10, 2017).

[2] Factories of the Future: Multi-annual roadmap for the contractual PPP under Horizon 2020. Technical report, EFFRA, 2013.

[3] Connected Manufacturing. Technical report, Industrial Value Chain Initiative, 2014.

[4] *Industrie 4.0 - Innovationen für die Produktion von morgen*, 2014. Bundesministerium für Bildung und Forschung.

[5] Strategic Investment Plan. Technical report, Digital Manufacturing and Design Innovation Institute, 2015.

[6] Business To Manufacturing Markup Language (B2MML). `https://isa-95.com/b2mml`, 2015 (accessed February 11, 2017).

[7] Esraa S Abdelall, Matthew C Frank, and Richard T Stone. Design for manufacturability-based feedback to mitigate design fixation. *Journal of Mechanical Design*, 140(9):091701, 2018.

[8] Yazen Alsafi and Valeriy Vyatkin. Ontology-based reconfiguration agent for intelligent mechatronic systems in flexible manufacturing. *Robotics and Computer-Integrated Manufacturing*, 26(4):381 – 391, 2010.

[9] Nikolas Antzoulatos, Elkin Castro, Lavindra de Silva, André Dionisio Rocha, Svetan Ratchev, and José Barata. A multi-agent framework for capability-based reconfiguration of industrial assembly systems. *International Journal of Production Research (IJPR)*, 55(10):2950–2960, 2017.

[10] Radu F Babiceanu and F Frank Chen. Development and Applications of Holonic Manufacturing Systems: A Survey. *Journal of Intelligent Manufacturing*, 17(1):111–131, 2006.

[11] Otto J Bakker, Jack C Chaplin, Lavindra de Silva, Paolo Felli, David Sanderson, Brian Logan, and Svetan Ratchev. Toward process control from formal models of transformable manufacturing systems. *Procedia CIRP*, 63:521–526, 2017.

[12] José Barata, Luís Camarinha-Matos, and Gonçalo Cíndido. A multiagent-based control system applied to an educational shop floor. *Robotics and Computer-Integrated Manufacturing*, 24(5):597 – 605, 2008.

[13] Conrad Bock and Michael Gruninger. PSL: A semantic domain for flow models. *Software & Systems Modeling*, 4(2):209–231, 2005.

[14] Miran Brezocnik and Joze Balic. A genetic-based approach to simulation of self-organizing assembly. *Robotics and Computer-Integrated Manufacturing*, 17(1):113 – 120, 2001.

[15] Gonçalo Cândido and José Barata. A multiagent control system for shop floor assembly. In *Proceedings of the international conference on Industrial Applications of Holonic and Multi-Agent Systems:Holonic and Multi-Agent Systems for Manufacturing*, pages 293–302, 2007.

[16] Jack C Chaplin, Otto J Bakker, Lavindra de Silva, David Sanderson, Emma Kelly, Brian Logan, and Svetan Ratchev. Evolvable Assembly Systems: A Distributed Architecture for Intelligent Manufacturing. *IFAC-PapersOnLine*, 48(3):2065–2070, 2015.

[17] Allan Cheng, Javier Esparza, and Jens Palsberg. Complexity results for 1-safe nets. *Theoretical Computer Science*, 147(1-2):117 – 136, 1995.

[18] Allen Cheng. *Reasoning About Concurrent Computational Systems*. PhD thesis, University of Aarhus, 1996.

[19] Andrei Ciortea, Simon Mayer, and Florian Michahelles. Repurposing manufacturing lines on the fly with multi-agent systems for the web of things. In *Proceedings of the International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 813–822, 2018.

[20] Lavindra de Silva, Paolo Felli, Jack C Chaplin, Brian Logan, David Sanderson, and Svetan Ratchev. Realisability of Production Recipes. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, pages 1449–1457, 2016.

[21] Lavindra de Silva, Paolo Felli, Jack C Chaplin, Brian Logan, David Sanderson, and Svetan Ratchev. Synthesising industry-standard manufacturing process controllers (demonstration). In *Proceedings of the International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 1811–1813, 2017.

[22] Hoda A ElMaraghy. Flexible and reconfigurable manufacturing systems paradigms. *International Journal of Flexible Manufacturing Systems*, 17(4):261–276, 2005.

[23] Dave Emerson, Haruhisa Kawamura, and Wayne Matthews. Plant-to-business (P2B) interoperability using the ISA-95 standard. *Yokogawa Technical Report*, 43:17, 2007.

[24] Kutluhan Erol, James Hendler, and Dana S Nau. HTN planning: Complexity and expressivity. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 1123–1128, 1994.

[25] Hakki Eskicioglu. The use of expert system building tools in process planning. *Engineering Applications of Artificial Intelligence*, 5(1):33 – 42, 1992.

[26] Javier Esparza and Mogens Nielsen. Decidability Issues for Petri Nets - a Survey. *Bulletin of the European Association for Theoretical Computer Science*, 52:245–262, 1994.

[27] Paolo Felli, Lavindra de Silva, Brian Logan, and Svetan Ratchev. Composite capabilities for cloud manufacturing. In *Proceedings of the International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 1809–1811, 2018.

[28] Martyn Fletcher, Duncan McFarlane, Andrew Lucas, James Brusey, and Dennis Jarvis. The cambridge packing cell — a holonic enterprise demonstrator. In Vladimír Mařík, Michal Pěchouček, and Jörg Müller, editors, *Multi-Agent Systems and Applications III*, pages 533–543. Springer Berlin Heidelberg, 2003.

[29] Regina Frei and Giovanna Di Marzo Serugendo. Self-organizing assembly systems. *IEEE Transactions on Systems, Man, and Cybernetics, Part C*, 41(6):885–897, 2011.

[30] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers Inc., 2004.

[31] David Golightly, David Sanderson, Paul Holmes, Svetan Ratchev, and Sarah Sharples. Design Requirements for Effective Hybrid Decision Making with Evolvable Assembly Systems. In *Proceedings of the European Conference on Cognitive Ergonomics (ECCE)*, pages 1–7, 2016.

[32] Muhammad Adnan Hashmi, Muhammd Usman Akram, and Amal El Fallah-Seghrouchni. Place: Planning based language for agents and computational environments. In Amal El Fallah-Seghrouchni, Alessandro Ricci, and Tran Cao Son, editors, *Engineering Multi-Agent Systems*, pages 142–158. Springer International Publishing, 2018.

[33] Erik Hollnagel and David D Woods. *Joint cognitive systems : foundations of cognitive systems engineering*. Taylor & Francis, 2005.

[34] Eeva Järvenpää. *Capability-based adaptation of production systems in a changing environment*. PhD Thesis, Julkaisu-Tampere University of Technology, 2012.

[35] Lalita Jategaonkar and Albert R Meyer. Deciding true concurrency equivalences on safe, finite nets. *Theoretical Computer Science*, 154(1):107 – 143, 1996.

[36] Albert T Jones, David Romero, and Thorsten Wuest. Modeling agents as joint cognitive systems in smart manufacturing systems. *Manufacturing Letters*, 17:6–8, 2018.

[37] Henning Kagermann, Johannes Helbig, Ariane Hellinger, and Wolfgang Wahlster. Recommendations for Implementing the Strategic Initiative INDUSTRIE 4.0: Securing the Future of German Manufacturing Industry; Final Report of the Industrie 4.0 Working Group. 2013.

[38] Yoram Koren, Uwe Heisel, Francesco Jovane, Toshimichi Moriwaki, Gumter Pritschow, Galip Ulsoy, and Hendrik Van Brussel. Reconfigurable Manufacturing Systems. *CIRP Annals - Manufacturing Technology*, 48(2):527–540, 1999.

[39] Paulo Leitão, Armando W Colombo, and Francisco Restivo. An approach to the formal specification of holonic control systems. In *Proceedings of the International Conference on Industrial Applications of Holonic and Multi-Agent Systems (HoloMAS)*, pages 59–70, 2003.

[40] Paulo Leitão, Vladimír Marík, and Pavel Vrba. Past, present, and future of industrial agent applications. *IEEE Transactions on Industrial Informatics*, 9(4):2360–2372, 2013.

[41] Paulo Leitão and Francisco Restivo. A holonic approach to dynamic manufacturing scheduling. *Robotics and Computer-Integrated Manufacturing*, 24(5):625 – 634, 2008.

[42] Hector J Levesque, Raymond Reiter, Yves Lesprance, Fangzhen Lin, and Richard B Scherl. Golog: A logic programming language for dynamic domains. *The Journal of Logic Programming*, 31(1):59 – 83, 1997.

[43] Jörg P Müller and Klaus Fischer. Application impact of multi-agent systems and technologies: A survey. In Onn Shehory and Arnon Sturm, editors, *Agent-Oriented Software Engineering: Reflections on Architectures, Methodologies, Languages, and Frameworks*, pages 27–53. Springer Berlin Heidelberg, 2014.

[44] Christian Müller-Schloer, Hartmut Schmeck, and Theo Ungerer, editors. *Organic Computing - A Paradigm Shift for Complex Systems*. Springer Basel, 2011.

[45] Pedro Neves and José Barata. Evolvable production systems. In *Proceedings of the IEEE International Symposium on Assembly and Manufacturing*, pages 189–195, 2009.

[46] Mauro Onori, José Barata, and Regina Frei. Evolvable Assembly Systems Basic Principles. In *International Conference on Information Technology For Balanced Manufacturing Systems*, volume 220 of *IFIP International Federation for Information Processing*, pages 317–328, 2006.

[47] Mauro Onori, Daniel Semere, and Bengt Lindberg. Evolvable systems: an approach to self-X production. *International Journal of Computer Integrated Manufacturing*, 24(5):506–516, 2011.

[48] Giovanni Di Orio, Gonçalo Cândido, and José Barata. The adapter module: A building block for self-learning production systems. *Robotics and Computer-Integrated Manufacturing*, 36:25 – 35, 2015.

[49] Anand S Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In *Proceedings of the European workshop on Modelling Autonomous Agents in a Multi-Agent World : agents breaking away*, pages 42–55, 1996.

[50] Jens Rasmussen, Annelise Mark Pejtersen, and Len P Goodstein. *Cognitive systems engineering*. Wiley New York, 1994.

[51] Mario Lucio Roloff, Marcelo Ricardo Stemmer, Jomi Fred Hübner, Robert Schmitt, Tilo Pfeifer, and Guido Huttemann. A multi-agent system for the production control of printed circuit boards using JaCaMo and Prometheus AEOlus. In *Proceedings of the IEEE International Conference on Industrial Informatics (INDIN)*, pages 236–241, 2014.

[52] David Sanderson, Nikolas Antzoulatos, Jack C Chaplin, Didac Busquets, Jeremy Pitt, Carl German, Alan Norbury, Emma Kelly, and Svetan Ratchev. Advanced Manufacturing: An Industrial Application for Collective Adaptive Systems. In *Proceedings of the IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW)*, pages 61–67, 2015.

[53] Bianca Scholten. *MES Guide for Executives: Why and how to Select, Implement, and Maintain a Manufacturing Execution System*. International Society of Automation, 2009.

[54] Manuel Silva and Enrique Teruel. Petri Nets for the design and operation of manufacturing systems. *European Journal of Control*, 3(3):182 – 199, 1997.

[55] Ingo Timm and Peer-Oliver Woelk. Ontology-based capability management for distributed problem solving in the manufacturing domain. In *Proceedings of the German Conference on Multiagent System Technologies (MATES)*, pages 168–179. 2003.

[56] John M Usher. Object oriented approach to feature-based process planning. In Hamid R Parsaei and M Jamshidi, editors, *Design and Implementation of Intelligent Manufacturing Systems*, pages 275–300. Prentice-Hall, Inc., 1995.

[57] Lihui Wang, Goran Adamson, Magnus Holm, and Philip Moore. A review of function blocks for process planning and control of manufacturing equipment. *Journal of Manufacturing Systems*, 31(3):269 – 279, 2012.

[58] M Kudret Yurtseven, Walter W Buchanan, and Melek Basak. Joint cognitive system design and process control. In *Proceedings of the Portland International Conference on Management of Engineering & Technology (PICMET)*, pages 2256–2262, 2009.

Appendix

In this section we list the algorithms that correspond to the flowcharts given in Figures 6, 7, and 8. Algorithms 2 and 3 were taken from [11] and slightly improved, e.g. to remove global variables.

Algorithm 1, when called with the following three parameters: the list **RES** of LTSs representing the assembly resources in the production system, the corresponding list of initial resource states **INITTOPSTATE**, and the empty set, returns the set of topology transitions **T**, which together with **INITTOPSTATE** forms the topology (see the EBNF syntax in Figure 3). This topology is one of the inputs for Algorithms 2 and 3.

Algorithm 1 ComputeTop

Input: The list of resources $\textbf{RESOURCE}_1, \ldots, \textbf{RESOURCE}_n$, denoted by **RES**, comprising the assembly system;
Their corresponding current states $\textbf{RESOURCESTATE}_1, \ldots, \textbf{RESOURCESTATE}_n$, denoted by **RESSTATES**;
A set of topology transitions **T**, which is initially the empty set.

Output: Set **T** is updated with the (valid) topology transitions that are possible from **RESSTATES**.

if **RESSTATES** was already visited **then**
  return **T**
end if
for each list **L** of resource transitions from **RESSTATES** **do** // Each $\textbf{RESOURCETRANS}_i$ in **L** is from $\textbf{RESOURCESTATE}_i$
  Let **TOPTRANS** be the topology transition (**RESSTATES**, **RESOPS**, **RESSTATESTO**) corresponding to **L**
  // **RESOPS** (resp. **RESSTATESTO**) combines **ASSEMBLYOP** (resp. **RESOURCESTATETO**) in each $\textbf{RESOURCETRANS}_i$
  if for each transfer operation *in* : x in **RESOPS** there exists exactly one *out* : x in **RESOPS**, and vice versa, **then**
    $\textbf{T} \leftarrow \textbf{T} \cup \{\textbf{TOPTRANS}\}$ // Topology transition **TOPTRANS** is valid, so add **TOPTRANS** to the set **T**
    $\textbf{T} \leftarrow$ ComputeTop(**RES**, **RESSTATESTO**, **T**) // Repeat all the above steps from **RESSTATESTO**
  end if
end for
return **T**

Algorithm 2 CheckManuf

Input: Topology **TOPOLOGY** and recipe **RECIPE**;
Current state of recipe **RECIPESTATE**;
Current states of all resources **RESSTATES**.

Output: Either *success* if **RECIPE** is manufacturable or *fail* otherwise.

Let **CURRENTPLAN** be the empty sequence
for each transition **RECIPETRANS** from **RECIPESTATE** **do** // Recall that **RECIPETRANS** is in **RECIPE**
  result $\leftarrow$ ProcessCompositeOp(**TOPOLOGY**, **RECIPE**, **RESSTATES**,
    COMPOSITEOP, **RECIPESTATETO**, **CURRENTPLAN**)
  // Recall that **COMPOSITEOP** and **RECIPESTATETO** are in **RECIPETRANS**
  if *result* = *fail* **then**
    return *fail*
  end if
end for
return *success*

Algorithm 3 ProcessCompositeOp

Input: Topology **TOPOLOGY** and recipe **RECIPE**;
Current states of all (part-annotated) resources **RESSTATES**;
Composite operation currently being processed **COMPOSITEOP**;
The next recipe state to explore **RECIPESTATETO**;
The current plan being composed **CURRPLAN**.
Output: Either *success* if **RECIPE** is manufacturable or *fail* otherwise.

1: **if** **COMPOSITEOP** has been completely processed **then**
2: Store **CURRPLAN** in the controller for the initial **COMPOSITEOP**
3: *result* ← CheckManuf(**TOPOLOGY**, **RECIPE**, **RECIPESTATETO**, **RESSTATES**)
4: **return** *result*
5: **end if**
6: **if** **RESSTATES** was already visited **then**
7: **return** *fail*
8: **end if**
9: **for** each transition **TOPTRANS** from **RESSTATES** **do** // **RESOPS** and **RESSTATESTO** are in **TOPTRANS** (in **TOPOLOGY**)
10: Let **FIRSTOP** be the first (non-guard) possibly parallel step of **COMPOSITEOP**
11: *result* ← *fail*
12: **RESPARTS2** ← Allocate(**FIRSTOP**, **RESOPS**, **RESPARTS**)
13: **if** **RESPARTS2** $\neq$ *false* **then**
14: **RESPARTS2** ← TransferParts(**RESPARTS2**, **RESOPS**)
15: **RESSTATESTO** ← AnnotateParts(**RESSTATESTO**, **RESPARTS2**)
16: **CURRPLAN** ← Append(**RESOPS**, **CURRPLAN**)
17: Let **REST** be the remaining steps of **COMPOSITEOP**
18: *result* ← ProcessCompositeOp(**TOPOLOGY**, **RECIPE**, **RESSTATESTO**, **REST**, **RECIPESTATETO**, **CURRPLAN**)
19: **end if**
20: **if** **RESOPS** has no observable operations **then** // If this holds, then **RESPARTS2** = *false*
21: **RESPARTS2** ← TransferParts(**RESPARTS**, **RESOPS**)
22: **RESSTATESTO** ← AnnotateParts(**RESSTATESTO**, **RESPARTS2**)
23: **CURRPLAN** ← Append(**RESOPS**, **CURRPLAN**)
24: *result* ← ProcessCompositeOp(**TOPOLOGY**, **RECIPE**, **RESSTATESTO**, **COMPOSITEOP**, **RECIPESTATETO**, **CURRPLAN**)
25: **end if**
26: **if** *result* = *success* **then**
27: **return** *result*
28: **end if**
29: **end for**
30: **return** *fail*