ECAI 2016
G.A. Kaminka et al. (Eds.)

doi:10.3233/978-1-61499-672-9-1449

Realisability of Production Recipes

Lavindra de Silva[1] and **Paolo Felli**[1] and **Jack C. Chaplin**[1] and
Brian Logan[2] and **David Sanderson**[1] and **Svetan Ratchev**[1]

Abstract. There is a rising demand for customised products with a high degree of complexity. To meet these demands, manufacturing lines are increasingly becoming autonomous, networked, and intelligent, with production lines being virtualised into a manufacturing cloud, and advertised either internally to a company, or externally in a public cloud. In this paper, we present a novel approach to two key problems in such future manufacturing systems: the *realisability problem* (whether a product can be manufactured by a set of manufacturing resources) and the *control problem* (how a particular product should be manufactured). We show how both production recipes specifying the steps necessary to manufacture a particular product, and manufacturing resources and their topology can be formalised as labelled transition systems, and define a novel simulation relation which captures what it means for a recipe to be realisable on a production topology. We show how a controller that can orchestrate the resources in order to manufacture the product on the topology can be extracted from the simulation relation, and give an algorithm to compute a simulation relation and a controller.

1 INTRODUCTION

Automating the manufacture of complex products is key to competitiveness in high-labour cost economies. However, automating the assembly of manufactured products presents significant challenges. It is widely acknowledged that responsiveness and customisability are key to the future of manufacturing [19], resulting in a drive towards *"batch-size-of-one"* production, in which each item produced differs from the items assembled immediately before and immediately after it [4]. In addition, there is drive towards *"manufacturing-as-a-service"* and the virtualisation and networking of resources to create *Cloud Manufacturing*, in which manufacturing software and resources are advertised and shared between members of a cloud [14]. In cloud environments, products may be manufactured by multiple cloud participants, representing multiple different enterprises connected via a supply chain. This trend toward flexible, adaptive, intelligent, and networked manufacturing systems has been termed the *fourth industrial revolution* or Industrie 4.0, in which decentralised intelligence in an *Internet of Things* connects embedded production resources to form *"smart factories"* that communicate and collaborate [12].

There is a growing body of work on automation to achieve flexibility, resilience, and monitoring in manufacturing. For example, *Flexible Manufacturing Systems* [7, 20, 10] increase the range of products that may be assembled, and *Reconfigurable Manufacturing Systems* [5, 13, 15, 21] reduce response time. However, to date, there has been little work on the assembly of highly-customised products in a highly-networked manufacturing environment.

In such a setting, the set of products that will be manufactured is not known in advance. Rather, production recipes specifying the steps necessary to manufacture a particular product are matched against virtualised manufacturing resources (such as a plant or assembly line), and the matched resource must then self-configure to assemble the product. Manufacturing control software must therefore make decisions both about *whether* a particular product can be manufactured by a particular set of manufacturing resources, and *how* a particular customised product should be manufactured by the resources (the steps an assembly line must perform in order to assemble the product). Whether a product can be manufactured by a set of manufacturing resources is termed the *realisability problem*; how it should be manufactured by those resources is termed the *control problem*.

In this paper, we present a new approach to the realisability and control problems in manufacturing. We show how both production recipes and manufacturing resources and their topology can be formalised as labelled transition systems, and define a novel simulation relation which captures what it means for a recipe to be realisable on a production topology. We give an algorithm for computing the simulation relation, and show how a controller to manufacture a product specified by a production recipe on a topology can be extracted from the simulation relation. To the best of our knowledge, our approach is the first fully-automated solution to the realisability and control problems in manufacturing. It forms a key component of the Evolvable Assembly Systems (EAS) architecture, an agent-based architecture for manufacturing control software designed to address rapidly changing product and process requirements including batch-size-of-one customised production [8]. The EAS architecture is being evaluated on a number of real-word production system demonstrators, and we illustrate our approach with a simple example of the manufacture of products (automotive hinges) by a flexible assembly platform.

2 ASSEMBLY SYSTEMS

In this section, we motivate and informally introduce the key ideas needed for the formal development, including production recipes, production resources, and their topology.

A *production recipe* specifies the steps necessary to manufacture a particular product, including the constituent parts, the tasks and associated parameters 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 manufacturing process, and how to respond to the results of tests (e.g., whether a partially completed product instance should be reworked or discarded following a test).

In industry, when recipe information is passed from the higher-

[1] Institute for Advanced Manufacturing, Faculty of Engineering, University of Nottingham, first.last@nottingham.ac.uk
[2] School of Computer Science, University of Nottingham, bsl@cs.nott.ac.uk

level enterprise control systems to lower level shop-floor control systems, it is usual for a *"recipe file"* to be used. Recipe files may be specified in a proprietary format, or the ANSI/ISA-95 (Enterprise-Control System Integration) family of standards [1] (also known as IEC/ISO 62264 [3]). One common approach to implementing the data models in the ISA-95 standard, is to use Business To Manufacturing Markup Language (B2MML) [2] XML schemas. Messages in B2MML contain an operation schedule, which specifies each operation to be performed. An operation consists of one or more requirements (e.g., for personnel, equipment, physical assets, or material), and precisely specifies the location and time the operation should be performed. While languages such as B2MML allow a *solution* to the control problem to be expressed (i.e., which resource should perform which operation on which part, and when), they do not provide an appropriate level of abstraction for specifying the *inputs* to the realisability and control problems. In our approach, we therefore specify recipies in a language that has some similarities to Hierarchical Task Networks (HTNs) [11] and Belief-Desire-Intention (BDI) [18] agent systems (see Section 3). For example, like these languages, we allow steps to be partially ordered (including interleaved), and do not specify which resource should perform each step.

A production recipe is enacted by a set of *production resources*. Each resource has a set of capabilities describing the basic actions it can perform, and in which order. Production resources are connected by transport links (conveyor belts, shuttles, manual item movement, etc.) to form a manufacturing line with a specific *production topology*. The production topology implicitly determines which production recipes can be manufactured by the line.

To illustrate these ideas, we briefly describe the Precision Assembly Demonstrator (PAD), a flexible assembly platform consisting of four production resources (see Figure 1a): two KUKA robot arms each with an associated workspace allowing the robots to place parts and perform operations, a testing and inspection workstation that can perform mechanical and vision-based tests, and a manual loading and unloading station where pallets of parts can be added or removed. There is additionally a shared, passive tool-changing rack between the two robot arms, and a shuttle transport system.

The shuttle transport system links workstations to give the production topology shown in Figure 1b. The shuttle transports a pallet carrier, which allows one pallet of parts at a time to be moved between resources. The robots use gripper end effectors to pick up or return pallets from the shuttle and place them on their respective workstations, where they can change end effectors from the tool rack and perform a variety of tasks.

The PAD demonstrator's primary production recipe family is a detent hinge for the use in automotive interiors. The simplest production recipe consists of a hollow plastic hinge consisting of two leaves (an interior and an exterior), which are linked with a metal hinge pin. More complex recipes are achieved by adjusting the hinge detent force by adding up to three metal balls and springs in slots in the interior hinge leaf. Each spring-ball pair increases the force required to engage the hinge. The robots assemble the hinge using a variety of gripper end effectors. A wide range of new end effectors may be added to alter the capabilities of the robot arms, such as new grippers for alternative hinge designs, glue applicators for securing the hinge pin, or engraving tools to give hinges serial codes. The flexibility of the PAD assembly platform results in both a realisability problem (how to determine if a recipe can be produced on a given set of hardware), and a control problem (how to manage the production of recipes on the hardware).

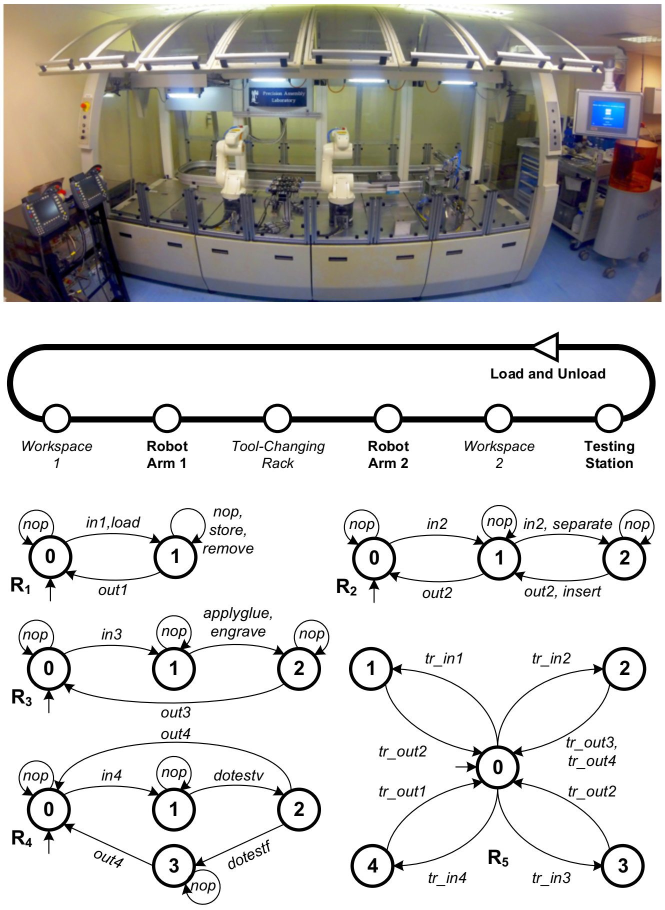

Figure 1: (a) Our Precision Assembly Demonstrator (top); (b) its diagrammatic layout (centre); and (c) its LTS model (bottom).

3 PRODUCTION RECIPE

In this section, we formalise production recipes as labelled transition systems (LTS), where labels are not just atomic steps but complex "task expressions".

We will make use of two sets: a set C of *part constants*, which represent parts used in production, and a set of *task labels*, or simply *tasks*, which represent operations. The latter set is the union of three mutually disjoint sets: the set of *observable* tasks T_{ob}, which only occur in recipes and non-transport resources; the set of *internal* tasks T_{in}, which only occur in transport resources; and the set of *synchronisation* tasks T_{syn}, which are used to "transfer" parts between production resources. Specifically, $T_{syn} = \{t^x_{\leftarrow} | x \in \mathbb{N}\} \cup \{t^x_{\rightarrow} | x \in \mathbb{N}\}$, namely, the set of *in* ($t^x_{\leftarrow}$) and *out* ($t^x_{\rightarrow}$) synchronisation tasks; parts are moved from resources performing *out* tasks (releasing a part) to corresponding resources performing *in* tasks (accepting a part). Further, we introduce the special task *nop* to denote the special "no-op" task, which represents idling.

Production recipes specify how (though not *where*) tasks and checks ought to be carried out in order to manufacture a desired end-product. Tasks are specified by *task expressions*.

The smallest task expression is of the form $t(\mathbf{c}, \mathbf{c}')$, and is called a *parameterised task* (*p-task*), where $t \in T_{ob}$, and $\mathbf{c} \in C^*$ and $\mathbf{c}' \in C^+$ are sequences of part constants such that any constant in C occurs at most once in $\mathbf{c}$ and in $\mathbf{c}'$. Given a p-task $t(\mathbf{c}, \mathbf{c}')$, sequences $\mathbf{c}$ and $\mathbf{c}'$ represent the "input" and "output" parameters of t, respectively: they represent the part(s) on which t is performed, and the possibly new part(s) that results from doing t_i. We use ϵ to denote the empty sequence, and denote $\mathbf{c}$ by ${}^{\circ}t$ and $\mathbf{c}'$ by t°. We sometimes write t instead of $t(\mathbf{c}, \mathbf{c}')$ when $\mathbf{c}$ and $\mathbf{c}'$ are not relevant.

For instance, the task $cut(c, c1 \cdot c2)$ represents an operation that takes a part c and produces two parts c_1 and c_2, whereas the task $load(\epsilon, c)$ represents an operation that introduces a new part c into the production facility.

The set of *general task expressions* is the smallest set of formulas, denoted by $Lang(\mathcal{T}_g)$, generated by the grammar $\mathcal{T}_g \ := ?\phi : \mathcal{T}$, where ϕ is a propositional formula from a propositional language P and $\mathcal{T}$ is a task expression. A *task expression* is a formula in the language generated by the grammar:

$$\mathcal{T} \ := \ t(\mathbf{c}, \mathbf{c}') \mid \mathcal{T}; \mathcal{T}' \mid \mathcal{T} \| \mathcal{T}' \mid \mathcal{T} \text{ “}|\text{” } \mathcal{T}'$$

The operator “;” denotes a sequence; “||” denotes parallel composition; and “|” denotes interleaved composition. We refer to a p-task or a parallel composition (of p-tasks) as an *atomic task expression* (or as *atomic*).

We impose two additional constraints on task expressions:

- any expression $\mathcal{T}_1 \| \ldots \| \mathcal{T}_m$ occurring in $\mathcal{T}$ is restricted such that each $\mathcal{T}_i$ is a p-task, and for each pair t_i, t_j of p-tasks, it does not hold that t_i and t_j mention the same part, and ${}^{\circ}t_i = {}^{\circ}t_j = \epsilon$;
- any interleaving $\mathcal{T}_1 | \ldots | \mathcal{T}_m$ occurring in $\mathcal{T}$ is restricted such that each $\mathcal{T}_i$ does not mention operator “|”.

The first constraint restricts parallelism to atomic tasks, and requires that parallel atomic tasks cannot share the same part constants, as a part can be a parameter of only one task at a time. It also forbids two or more parallel tasks with empty input parameters. The second constraint forbids “nested” interleaving.

The general task expression $?\phi : \mathcal{T}$ specifies that $\mathcal{T}$ can only happen if the guard ϕ holds, based on a valuation for ϕ that is available at runtime (based on data collected in real-time). Since our analysis is instead performed offline, in the remainder of this paper, we will ignore any guard appearing in general task expressions, thus considering any valuation as equally possible. Nonetheless, guards in general task expressions stress the fact that any formula in the language of task expressions $Lang(\mathcal{T})$ may be associated with a guard.

With these definitions at hand, it is now possible to define our notion of *production recipe* (or simply recipe). A *recipe* is a labelled transition system in which labels are task expressions and nodes represent the states of parts in the assembly. This definition essentially allows for loops in task expressions, and for the specification of alternatives among task expressions.

Definition 1 (Recipe) *A recipe is a tuple $\boldsymbol{R} = (s^0, S, L, \rightarrow)$, where S is a finite set of states, $s^0 \in S$ is the initial state, $L \subseteq Lang(\mathcal{T})$ is a set of task expressions, and $\rightarrow : S \times L \mapsto S$ is a non-empty transition function. We denote a transition from state s to s', with task expression $\mathcal{T}$, either by $s \xrightarrow{\mathcal{T}} s'$ or $(s, \mathcal{T}, s') \in \rightarrow$.*

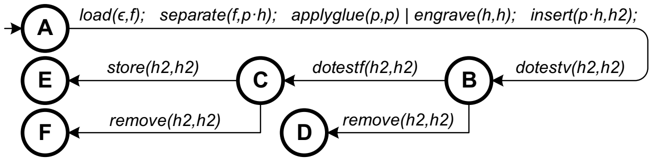
A
B
C
D
E
F
load(ϵ,f); separate(f,p·h); applyglue(p,p) | engrave(h,h); insert(p·h,h2);
dotestv(h2,h2)
dotestf(h2,h2)
store(h2,h2)
remove(h2,h2)
remove(h2,h2)

Figure 2: An example of a recipe. For readability, the sequence labelling the edge between A and B has been separated.

Figure 2 shows an example of a recipe to be executed on the assembly platform in Figure 1. The first two p-tasks request a new pallet fixture (f) to be loaded and then separated into the hinge pin (p) and hollow hinge (h). These are followed by an interleaved composition, which requires glue to be applied to p, and for h to be engraved with a serial number (in any order). The two parts are then combined to form the final hinge ($h2$), by inserting p into h. The next step requests a visual test to be performed on $h2$, which simply collects test data. After this, the recipe either requests a force test, or for the hinge to be removed, based on the outcome of evaluating guards (not shown) against real-time visual-test data. Similarly, the outcome of evaluating guards against real-time force-test data determines whether the hinge is recycled or stored for delivery.

A *trace* of a recipe $\boldsymbol{R}$ is a sequence $\pi = s_0 \xrightarrow{\mathcal{T}_1} s_1 \xrightarrow{\mathcal{T}_2} \cdots \xrightarrow{\mathcal{T}_n} s_n$, namely, a sequence of states and transitions, of arbitrary length $n \geq 0$, such that $s_{i-1} \xrightarrow{\mathcal{T}_i} s_i$ for each $i \in [1, n]$. If not specified, we assume $s_0 = s^0$.

Finally, we assume that cycles in a recipe are bounded, in order to ensure that manufacturing the end-product will eventually cease. Since bounded cycles can easily be removed by unfolding the recipe graph, in what follows we deal only with acyclic recipes, i.e., recipes where the edge-labelled graph $(S, \rightarrow)$ is acyclic.

4 PRODUCTION RESOURCE & TOPOLOGY

A *production resource* represents a production “station” (or “cell”), which performs observable operations on parts. A *transport resource* represents a transport mechanism such as a conveyer belt, which performs internal operations that route parts between stations. We model production and transport resources as labelled transition systems, where a node represents a state of the resource, and an outgoing edge from a node corresponds to a task that can be performed from that node.

Definition 2 (Resource) *A production (resp. transport) resource is a tuple $(\underline{s}^0, \underline{S}, \underline{L}, \Longrightarrow)$, where $\underline{S}$ is a set of states, $\underline{s}^0 \in \underline{S}$ is the initial state, $\underline{L} \subseteq T_{ob} \cup T_{syn} \cup \{nop\}$ (resp. $\underline{L} \subseteq T_{in} \cup T_{syn} \cup \{nop\}$) is a set of tasks,[3] and $\Longrightarrow : \underline{S} \times \underline{L} \mapsto \underline{S}$ is a non-empty transition function.*

Intuitively, the *nop* task allows a resource to idle for either a fixed or an unbounded number of steps, before and/or after performing other tasks. Note that $\Longrightarrow$ is a function: we assume that when a resource performs a task from a state, there exists only one possible successor state for that task.

Figure 1c shows a simplified model of the resources in Figure 1. Multiple transitions between two states are depicted as one, with labels separated by a comma. Resource R_1 can load a new pallet (containing a hollow hinge and hinge pin), and also store a hinge for removal or delivery. Labels *in1* and *out1* are *in* and *out* synchronisation tasks, respectively. Resource R_2 is the first robotic arm in Figure 1, which can either accept (via *in2*) a pallet fixture and separate it into the hollow hinge and pin, or accept these two parts, one after the other, and then insert the pin into the hinge, thereby again producing a single part. The second robotic arm is modelled as R_3, which can either apply glue to a given part, or engrave it with a serial number. Resource R_4 is the testing station, which can first analyse a hinge to gather visual data about its assembly, followed possibly by

[3] We could also allow production resources to specify internal tasks, which would then enable operations such as data logging (e.g. from sensors).

an analysis to gather data about hinge force. The latter is not allowed before doing the former because a wrongly assembled hinge could become damaged if it is manipulated to gather force data. Moreover, due to design constraints, the resource also cannot idle between these two operations, represented by the lack of a *nop* transition.

Finally, R_5 is a transport resource, modelling the shuttle system in Figure 1. This resource represents the "legal routes" between production resources: e.g., while a newly loaded fixture can be delivered from R_1 to R_2, delivering a glued part from R_3 to R_4 is forbidden, to prevent the part from becoming affixed during force analysis. Thus, intuitively, "out" synchronisations in production resources correspond to "in" synchronisations in transport resources, and vice versa. For example, $out1$ in R_1 corresponds to tr_in1 in R_5, and tr_out2 in R_5 corresponds to $in2$ in R_2.

A *production topology* (or simply *topology*) "connects" the available resources via synchronisation tasks, and represents the layout of the production system. Technically, a topology is the cross product of elements in resource tuples, excluding "invalid" transitions, i.e., those having an *out* synchronisation without the corresponding *in* synchronisation (and vice versa).

In what follows, we use two auxiliary notions. First, for any pair of states $\underline{s}, \underline{s}' \in \underline{S}$ and label $\mathbf{t} \in \underline{L}$, we use $\underline{s} \xRightarrow{\mathbf{t}} \underline{s}'$ to denote $(\underline{s}, \mathbf{t}, \underline{s}') \in \Longrightarrow$. Second, we define the *complement* of a synchronisation task $t \in T_{syn}$, denoted by $\sim t$, as $t^x_\leftarrow$ if $t = t^x_\rightarrow$, and as $t^x_\rightarrow$ otherwise, where $x \in \mathbb{N}$. For instance, referring to Figure 1c, if $t = out1$, then $\sim t = tr\_in1$. While, strictly speaking, the definition requires $\sim t = in1$, in the figure we have used the prefix "*tr*" for the purpose of readability. Taking inspiration from [6], we define a production topology as follows.

Definition 3 (Topology) *Let $R_1, \ldots, R_n$ be resources, where each $R_i = (\underline{s}^0_i, \underline{S}_i, \underline{L}_i, \Longrightarrow_i)$. A* <u>*topology*</u> *is a tuple $(\underline{s}^0, \underline{S}, \underline{L}, \Longrightarrow)$, where $\underline{S} = \underline{S}_1 \times \cdots \times \underline{S}_n$ is the set of states; $\underline{s}^0 = (\underline{s}^0_1, \ldots, \underline{s}^0_n)$ is the initial state; $\underline{L} = \underline{L}_1 \times \cdots \times \underline{L}_n$ is the set of concurrent tasks; and the transition relation $\Longrightarrow$ is such that for any $\underline{s}, \underline{s}' \in \underline{S}$ and $\mathbf{t} \in \underline{L}$, we have $\underline{s} \xRightarrow{\mathbf{t}} \underline{s}'$ iff for all $i \in [1, n]$:*[4]

1. *$t_i \notin T_{syn}$ and $\underline{s}_i \xRightarrow{t_i}_i \underline{s}'_i$; or*
2. *$t_i \in T_{syn}$ and $\underline{s}_i \xRightarrow{t_i}_i \underline{s}'_i$, and there exists exactly one $j \in [1, n]$ such that $\underline{s}_j \xRightarrow{t_j}_j \underline{s}'_j$ and $t_j = \sim t_i$.*

Thus, the topology depicts the concurrent execution of tasks t_i in different resources. Without loss of generality, condition (2) checks that, within a single transition, a particular synchronisation only takes place between one resource and exactly one other resource. This is because we later use synchronisations to unambiguously transfer parts between resources.

5 REALISABILITY OF A RECIPE ON A TOPOLOGY

We shall now define what it means for a recipe to be realisable on a topology. Our definition relies on some auxiliary notions, particularly involving the movement of parts and the execution of a recipe on a topology.

The notion of a topology as defined above is "static" in that it does not account for parts which are moved between resources and manipulated by them. Thus, given a topology $(\underline{s}^0, \underline{S}, \underline{L}, \Longrightarrow)$, we keep track of the movement of parts (constants) during production via a <u>resource vector</u> $\mathbf{r} = (\mathbf{c}_1, \ldots, \mathbf{c}_n)$, where each $\mathbf{c}_i \in C^*$ is a (possibly empty) sequence of parts that do not occur anywhere else in $\mathbf{r}$; we denote $\mathbf{c}_i$ by $\mathbf{r}(i)$ for any $i \in [1, n]$, and the set of all possible resource vectors as V. Intuitively, we associate each state in the topology with a resource vector $\mathbf{r}$, specifying which parts are currently allocated to each resource. Note that each element in vector $\mathbf{r}$ is a sequence and not a set: we assume a first-in-first-out approach when moving parts between resources.

Part constants get allocated to resources as p-tasks in recipes are processed. A resource R_j currently in state $\underline{s}_j$ can execute an (atomic) p-task $t(\mathbf{c}, \mathbf{c}')$ only if *(a)* the task t is available from state $\underline{s}_j$ in R_j, and *(b)* the parts $^\circ t$ appearing as the input parameters of t are currently allocated to the resource, namely, $\mathbf{r}(j) = \mathbf{c}$. After its execution, parts $\mathbf{c}'$ appearing as its output parameters (i.e., t°) are allocated to R_j. For example, the $separate(f, p \cdot h)$ p-task in Figure 2 is executable on resource R_2 in Figure 1c only if part f is currently allocated to R_2, and the resource is in state 1. Executing the task results in f being "removed" and R_2 being allocated parts p and h.

Formally, given a task expression $\mathcal{T} = t_1 \parallel \ldots \parallel t_m$, a resource vector $\mathbf{r}$, a state $\underline{s}$, and a transition $\underline{s} \xRightarrow{\mathbf{t}} \underline{s}'$ with $\mathbf{t} = (t'_1, \ldots, t'_n)$, a resource vector $\mathbf{r}'$ is an *allocation* of $\mathcal{T}$ to $\mathbf{t}$ with respect to $\mathbf{r}$, denoted $\mathbf{r}' \in \text{AL}(\mathbf{r}, \mathcal{T}, \mathbf{t})$, if and only if

- $\forall j \in [1, n]$, either $t'_j \notin T_{ob}$ and $\mathbf{r}'(j) = \mathbf{r}(j)$, or $\exists i \in [1, m]$ s.t.

$$\textit{(a)}\ t'_j = t_i \qquad \textit{(b)}\ \mathbf{r}(j) = {}^\circ t_i \text{ and } \mathbf{r}'(j) = t_i^\circ$$

- $\forall i \in [1, m], \exists j \in [1, n]$ s.t. *(a)* and *(b)*.

In other words, there must exist a one-to-one mapping from p-tasks t_i in $\mathcal{T}$ to observable tasks t'_j in transition $\mathbf{t}$, and from the parts associated with t_i to the ones corresponding to t'_j (i.e., $\mathbf{r}(j)$).

Allocated parts are transferred across resources by synchronisation tasks. Formally, a resource vector $\mathbf{r}'$ is a *transfer* or "move" of parts from $\mathbf{r}$ relative to $\mathbf{t}$ ($\mathbf{r}$ and $\mathbf{t}$ are as above), denoted $\mathbf{r}' = \text{MOV}(\mathbf{r}, \mathbf{t})$, if $\mathbf{r}'$ is obtained from $\mathbf{r}$ by replacing each $\mathbf{c}'_i, \mathbf{c}'_j \in \mathbf{r}$ ($\mathbf{c}'_i = c_1 \cdots c_m$) with respectively $c_2 \cdots c_m$ and $\mathbf{c}'_j \cdot c_1$, only if $\mathbf{t}(i) = t^x_\rightarrow$, $\mathbf{t}(j) = t^x_\leftarrow$ ($x \in \mathbb{N}$), and $m > 0$; if $m = 0$, then $\mathbf{r}' = \mathbf{r}$.[5] For example, the part p in resource R_2 in Figure 1c is moved to resource R_5 via the synchronisation involving tasks $out2$ and tr_in2.

Observe that a resource vector encodes the allocation of parts to resources (that is, which parts are currently allocated to each resource), and part allocations are not considered in the description of the resource itself. In other words, the description of a resource is purely behavioural, in the sense that it encodes the sequence of operations that it can (or is allowed to) execute, which is not constrained by the presence of a part in a given internal state as in the case of, e.g., Petri Nets [17]. The size of a resource vector is thus equal to the number of resources, and the allocation of parts is independent from the current state of the topology. This is an essential point to guarantee our complexity result.

The final auxiliary notion needed to define the execution of a recipe on a topology is the standard notion of a *linearisation*, which we have adapted for interleaved compositions. Formally, a *linearisation* $\mathcal{T}'$ of a task expression $\mathcal{T} = \mathcal{T}_1|\ldots|\mathcal{T}_m$, denoted $\mathcal{T}' \in \text{LIN}(\mathcal{T})$, is defined inductively as follows. If $m \in \{1, 2\}$, then $\mathcal{T}'$ is any sequence of the form $\mathcal{T}_1^1; \mathcal{T}_2^1; \mathcal{T}_1^2; \mathcal{T}_2^2; \ldots; \mathcal{T}_1^n; \mathcal{T}_2^n$, where $\mathcal{T}_1 = \mathcal{T}_1^1; \ldots; \mathcal{T}_1^n, \mathcal{T}_2 = \mathcal{T}_2^1; \ldots; \mathcal{T}_2^n$, and each $\mathcal{T}_i^j$ is a possibly empty sequence. If $m > 2$, then

$$\mathcal{T}' \in \bigcup_{\mathcal{T}'' \in \text{LIN}(\mathcal{T}_2|\ldots|\mathcal{T}_m)} \text{LIN}(\mathcal{T}_1|\mathcal{T}'').$$

[4] Recall that $\underline{s} = (\underline{s}_1, \ldots, \underline{s}_n)$, $\underline{s}' = (\underline{s}'_1, \ldots, \underline{s}'_n)$ and $\mathbf{t} = (t_1, \ldots, t_n)$.

[5] Given a label $\mathbf{t} = (t_1, \ldots, t_k)$, we denote t_i by $\mathbf{t}(i)$, for any $i \in [1, k]$.

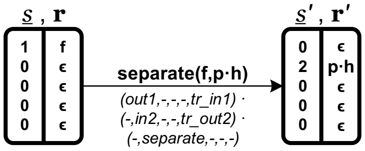

Figure 3: A graphical representation of a recursive application of the $\textsc{nxt}$ operator. Given the couple $(\underline{s}, \mathbf{r})$, where $\underline{s} = (1, 0, 0, 0, 0)$ and $\mathbf{r} = (f, \epsilon, \epsilon, \epsilon, \epsilon)$, the state $(\underline{s}', \mathbf{r}')$ is in $\textsc{nxt}((\underline{s}, \mathbf{r}), \mathit{separate}(f, p \cdot h))$ due to two unobservable transitions (part f is moved to resource R_2 via two synchronisation steps), thereby allocating the p-task to R_2.

That is, when given two interleaved task expressions $\mathcal{T}_1$ and $\mathcal{T}_2$, we make them the same length by arbitrarily "padding" with empty elements, and then merging the two resulting expressions to form one possible $\mathcal{T}'$.

We can now define what it means for a recipe to be executable on a topology. We do this with the following operator, which captures what it means to "execute" an arbitrary task expression on the topology $\boldsymbol{P}$, given a state $\underline{s}$ and resource vector $\mathbf{r}$. Intuitively, it returns the set of possible successor states of the topology, together with the corresponding next resource vectors.

In what follows, let $\sigma = (\underline{s}, \mathbf{r})$.

Definition 4 ($\textsc{nxt}$) *We define the operator* $\textsc{nxt}(\sigma, \mathcal{T}) =$

$$\begin{cases} \bigcup\limits_{\sigma' \in \Sigma} \textsc{nxt}(\sigma', \mathcal{T}_2) & \textit{if } \mathcal{T} = \mathcal{T}_1; \mathcal{T}_2 \textit{ and } \Sigma = \textsc{nxt}(\sigma, \mathcal{T}_1); \\[2ex] \bigcup\limits_{\mathcal{T}' \in \textsc{lin}(\mathcal{T})} \textsc{nxt}(\sigma, \mathcal{T}') & \textit{if } \mathcal{T} = \mathcal{T}_1 | \ldots | \mathcal{T}_n, n > 1; \\[2ex] \Sigma' \bigcup\limits_{\sigma' \in \Sigma} \textsc{nxt}(\sigma', \mathcal{T}) & \textit{if } \mathcal{T} = t_1 \| \ldots \| t_n, n \geq 1, \\ \qquad\qquad \textbf{\textit{(4.1)}} & \Sigma' = \{(\underline{s}', \mathbf{r}') \mid \underline{s} \overset{\mathbf{t}}{\Longrightarrow} \underline{s}', \mathbf{t} \in \underline{L}, \\ & \mathbf{r}'' \in \textsc{al}(\mathbf{r}, \mathcal{T}, \mathbf{t}), \mathbf{r}' = \textsc{mov}(\mathbf{r}'', \mathbf{t})\}, \textit{ and} \\[2ex] \qquad\qquad \textbf{\textit{(4.2)}} & \Sigma = \{(\underline{s}', \mathbf{r}') \mid \underline{s} \overset{\mathbf{t}}{\Longrightarrow} \underline{s}', \mathbf{t} \in \underline{L}, \\ & \mathbf{r}' = \textsc{mov}(\mathbf{r}, \mathbf{t}), \forall t \in \mathbf{t}, t \notin T_{ob}\}. \end{cases}$$

In this definition, an element $(\underline{s}', \mathbf{r}') \in \textsc{nxt}((\underline{s}, \mathbf{r}), \mathcal{T})$ is a "successor" of $(\underline{s}, \mathbf{r})$, where $\underline{s}'$ in the topology is a final state reachable from $\underline{s}$ under guidance from $\mathcal{T}$, and $\mathbf{r}'$ is the resource vector of $\underline{s}'$. More specifically,

- if $\mathcal{T} = \mathcal{T}_1; \mathcal{T}_2$, then one task at a time is considered according to the sequence;
- if $\mathcal{T} = \mathcal{T}_1 | \ldots | \mathcal{T}_n$, then all linearisations of $\mathcal{T}$ are considered;
- if $\mathcal{T} = t_1 \parallel \ldots \parallel t_n$ (including the base case $\mathcal{T} = t_1$), then Σ' is the set of couples $(\underline{s}', \mathbf{r}')$ such that $\underline{s}'$ is a successor of $\underline{s}$ (in the topology) for some label $\mathbf{t}$, and $\mathbf{r}'$ is the new resource vector obtained from $\mathbf{r}$ by first allocating $\mathcal{T}$ to $\mathbf{t}$, and then moving parts according to synchronisation tasks in $\mathbf{t}$. Similarly, Σ is the set of couples $(\underline{s}', \mathbf{r}')$ obtained from $(\underline{s}, \mathbf{r})$ when $\mathbf{t}$ is an "unobservable" transition (one in which no observable tasks occur).

Thus, $\textsc{nxt}((\underline{s}, \mathbf{r}), \mathcal{T})$ is the set of couples $(\underline{s}', \mathbf{r}')$, where $\underline{s}'$ is reachable from $\underline{s}$ by following a possibly empty sequence of unobservable transitions, followed by an allocation of $\mathcal{T}$, and $\mathbf{r}'$ is the resulting resource vector. Figure 3 depicts one application of the $\textsc{nxt}$ operator in the context of Figure 1c.

We extend the above notion of executability to traces of a recipe $\boldsymbol{R}$ as follows. A trace $\pi = s_0 \xrightarrow{\mathcal{T}_1} s_1 \xrightarrow{\mathcal{T}_2} \cdots \xrightarrow{\mathcal{T}_n} s_n$ of $\boldsymbol{R}$ is realisable in a topology $\boldsymbol{P}$ iff *(a)* there exists a sequence of n couples $(\underline{s}_1, \mathbf{r}_1) \cdots (\underline{s}_n, \mathbf{r}_n)$ such that $(\underline{s}_j, \mathbf{r}_j) \in \textsc{nxt}((\underline{s}_{j-1}, \mathbf{r}_{j-1}), \mathcal{T}_j)$ for each $j \in [1, n]$, and *(b)* $(\underline{s}_0, \mathbf{r}_0) = (\underline{s}^0, \mathbf{r}^0)$. For brevity, we shall write $(\underline{s}_n, \mathbf{r}_n) \in \textsc{nxt}^*((\underline{s}_0, \mathbf{r}_0), \pi)$. Then, we say that $\boldsymbol{R}$ is realisable in $\boldsymbol{P}$ iff any trace π of $\boldsymbol{R}$ (recall that their number is finite) is realisable in $\boldsymbol{P}$ from its initial state, i.e., $\exists (\underline{s}', \mathbf{r}') \in \textsc{nxt}^*((\underline{s}^0, \mathbf{r}^0), \pi)$ for any trace π of $\boldsymbol{R}$.

Finally, note that the definition of realisability highlights how our notion of a recipe is flexible. Multiple branches existing from a given state constitute a universal requirement (AND-nodes), as we need to be sure that, at run-time, *any* possible trace can be realised. At the same time, task expressions with interleaving represent existential requirements (OR-nodes), as it is sufficient to find *a possible* realisable linearisation. This makes the control problem in our setting more involved than considering any possible trace of a finite-state machine labelled with atomic tasks (as is the case in Behaviour Composition [9]), or the language expressing the set of legal traces of the system (as for Discrete Event Systems [22]).

5.1 Checking Realisability via Simulation

We can now define the notion of a *task simulation relation* for a topology and recipe, inspired by the mathematical notion of simulation [16]. Although this notion is generally applied to infinite processes, its application here to finite recipes allows us to capture the realisability property with respect to the state of the manufacturing facility. Intuitively, it relates states in the topology to states in the recipe with respect to current resource allocations, such that each transition (task expression) in the recipe can be executed by transitions (tasks) in the topology, and the same holds iteratively for the entire recipe, irrespective of the actual recipe trace that might be followed at execution-time (which depends on the outcome of evaluating guards in the recipe).

Definition 5 (Simulation) *Let* $\boldsymbol{P} = (\underline{s}^0, \underline{S}, \underline{L}, \Longrightarrow)$ *be a topology, and* $\boldsymbol{R} = (s^0, S, L, \rightarrow)$ *a recipe. A task simulation relation is a relation* $\textsc{sim} \subseteq \underline{S} \times S \times V$,[6] *such that a tuple* $(\underline{s}, s, \mathbf{r}) \in \textsc{sim}$ *implies that for any* $\mathcal{T}$ *and* s', *if* $s \xrightarrow{\mathcal{T}} s'$ *then there exists a state* $\underline{s}'$ *and a resource vector* $\mathbf{r}'$ *such that* $(\underline{s}', s', \mathbf{r}') \in \textsc{sim}$, *with* $(\underline{s}', \mathbf{r}') \in \textsc{nxt}((\underline{s}, \mathbf{r}), \mathcal{T})$.

We say that a state $\underline{s} \in \underline{S}$ simulates a state $s \in S$ with respect to a resource vector $\mathbf{r}$ if and only if there exists a task simulation relation $\textsc{sim}$ such that $(\underline{s}, s, \mathbf{r}) \in \textsc{sim}$. Indeed, many such task simulation relations may exist, each accounting for one or more ways of realising the recipe. Finally, topology $\boldsymbol{P}$ simulates recipe $\boldsymbol{R}$ if and only if there exists a task simulation relation $\textsc{sim}$ (one is sufficient) such that $(\underline{s}^0, s^0, \mathbf{r}^0) \in \textsc{sim}$, where $\mathbf{r}^0$ denotes the resource vector (with $|\mathbf{r}^0| = |\underline{s}^0|$) composed of empty sequences, namely, the "empty" resource vector.

Theorem 1 (Realisability via simulation) *Given a topology* $\boldsymbol{P} = (\underline{s}^0, \underline{S}, \underline{L}, \Longrightarrow)$ *and recipe* $\boldsymbol{R} = (s^0, S, L, \rightarrow)$, *the recipe is realisable in* $\boldsymbol{P}$ *iff* $\boldsymbol{P}$ *simulates* $\boldsymbol{R}$.

The proof for this is straightforward, as a task simulation only expresses an *invariant* property with respect to task allocation (the $\textsc{nxt}$ operator). Proceeding by contradiction, suppose $\boldsymbol{R}$ is realisable in $\boldsymbol{P}$ but that $\boldsymbol{P}$ does not simulate $\boldsymbol{R}$. By the definition of a task simulation

[6] Recall that V is the set of all resource vectors.

relation, this means that there exists a trace π in $\boldsymbol{R}$ such that for any $(\underline{s}', \mathbf{r}') \in \text{NXT}^*((\underline{s}^0, \mathbf{r}^0), \pi)$ we have that $(\underline{s}', s', \mathbf{r}') \notin \text{SIM}$, with s' being the last state in π. Hence, it can be seen that π (and thus $\boldsymbol{R}$) is not realisable in $\boldsymbol{P}$. The proof for the opposite direction is similar.

We conclude this section by observing that our notion of task simulation has some similarity with the notion of simulation between transition systems that has been investigated to solve the problem of Behaviour Composition [9]. However there are some fundamental differences. First, the task simulation relation is defined with respect to complex task expressions that can be allocated to resources in parallel, while the notion of a simulation relation applied to behaviour composition assumes that only one behaviour module at the time is allowed to execute actions. Second, each task in the recipe may here be realised by a sequence of observable and unobservable transitions in the topology, which is a more complex setting than the other.

6 CONTROLLER SYNTHESIS

When it is possible to manufacture a product using the given set of resources, that is, when there exists a task simulation relation between the recipe and the topology, the next step is to *synthesise* a *controller* able to orchestrate the resources in order to execute the recipe and thus manufacture the product.

Crucially, the task simulation relation between a topology $\boldsymbol{P}$ and recipe $\boldsymbol{R}$ alone does not hold all the information that is necessary in order to extract solutions: the relation is sufficient to answer whether we *can* orchestrate the resources in order to realise the recipe, but not *how*. For instance, given a tuple $(\underline{s}, s, \mathbf{r})$ in the task simulation relation, we know, when considering a transition $s\xrightarrow{\mathcal{T}_1|\mathcal{T}_2}s'$ in the recipe, that there must exist a linearisation $\text{LIN}(\mathcal{T}_1|\mathcal{T}_2)$ that is executable on the topology, but we do not know *which*. Indeed, such information is not stored in the relation. The same applies to (atomic) p-tasks: additional, unobservable transitions may be needed before the p-task can be executed, which the relation does not keep track of.

This additional information is the set of sequences of transitions $(\mathbf{t}_1 \cdots \mathbf{t}_k)$ in the topology that realise a task expression $\mathcal{T}$, for any couple $\sigma = (\underline{s}, \mathbf{r})$ and $\sigma' \in \text{NXT}(\sigma, \mathcal{T})$, where $\sigma' = (\underline{s}', \mathbf{r}')$. We denote this set by $\text{TRANSOF}(\sigma, \mathcal{T}, \sigma')$. It is defined similarly to the operator $\text{NXT}(\sigma, \mathcal{T})$ in Definition 4: an element $(\mathbf{t}_1 \cdots \mathbf{t}_k)$ is in the set $\text{TRANSOF}(\sigma, \mathcal{T}, \sigma')$ if and only it is one of the sequences of vectors $\mathbf{t}$ corresponding to k recursive applications of steps (4.1) and (4.2) in the definition. Thus, if $(\mathbf{t}_1 \cdots \mathbf{t}_k) \in \text{TRANSOF}(\sigma, \mathcal{T}, \sigma')$, then the trace

$$\underline{s} \xRightarrow{\mathbf{t}_1} \cdots \xRightarrow{\mathbf{t}_k} \underline{s}'$$

is a trace of $\boldsymbol{P}$. For example, the following sequence of transitions $(out1, t, t, t, tr\_in1) \cdot (t, in2, t, t, tr\_out2) \cdot (t, separate, t, t, t)$, with $t = nop$, is in $\text{TRANSOF}(\sigma, \mathcal{T}, \sigma')$, for σ and σ' as in Figure 3. We omit the full definition for brevity, but we make its steps clearer with an algorithm.

Let $\boldsymbol{P}$ be a production topology and $\boldsymbol{R}$ a production recipe. Suppose that $\boldsymbol{P}$ simulates $\boldsymbol{R}$, i.e., there exists a task simulation relation SIM between $\boldsymbol{P}$ and $\boldsymbol{R}$. Then, we define a "controller" as a state machine in which a transition encodes a sequence of topology transitions that ought to be executed, given a transition in the recipe.

Definition 6 (Controller) *Let* $\boldsymbol{P} = (\underline{s}^0, \underline{S}, \underline{L}, \Longrightarrow)$ *and* $\boldsymbol{R} = (s^0, S, L, \rightarrow)$ *be as above. A* *controller* *for* $\boldsymbol{R}$ *and* $\boldsymbol{P}$ *is a finite state machine* $\boldsymbol{C} = (\text{SIM}, \delta)$*, where*

- SIM *is a non empty task simulation relation, whose elements correspond to the set of states in the controller;*
- $\delta : \text{SIM} \times Lang(\mathcal{T}) \times \underline{L}^+ \mapsto \text{SIM}$ *is a transition function: given a state, an arbitrary task expression and a sequence of transitions in the topology, it returns the successor state. Specifically, for any state* $(\underline{s}', s', \mathbf{r}')$ *and task expression* $\mathcal{T}$*, if and only if* $s\xrightarrow{\mathcal{T}}s'$ *for some* s'*, then there exists at least one transition* $(\underline{s}', s', \mathbf{r}') = \delta((\underline{s}, s, \mathbf{r}), \mathcal{T}, (\mathbf{t}_1 \cdots \mathbf{t}_k))$*, with* $(\underline{s}', \mathbf{r}') \in \text{NXT}((\underline{s}, \mathbf{r}), \mathcal{T})$*, and* $(\mathbf{t}_1 \cdots \mathbf{t}_k) \in \text{TRANSOF}((\underline{s}, \mathbf{r}), \mathcal{T}, (\underline{s}', \mathbf{r}'))$.

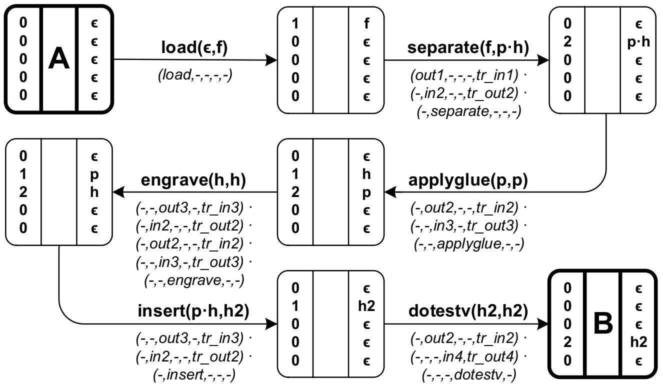

Figure 4: A graphical representation of a single transition in a controller between two tuples $(\underline{s}, s, \mathbf{r}) \in \text{SIM}$ (the first and the last), and therefore states in a possible controller. The states not in bold correspond to "intermediate" steps, namely, applications of the NXT operator to atomic tasks. For instance, the second transition corresponds to $(\underline{s}', s') \in \text{NXT}((\underline{s}, s), separate(f, p \cdot h))$, as in Figure 3. Therefore, the transition in the controller is labelled with *(a)* the entire task expression $\mathcal{T}$ between A and B as in Figure 2, and *(b)* the complete sequence of transitions in the topology that we need to realise $\mathcal{T}$, as they appear in this figure. Dashes are used in place of *nop* tasks for readability.

Figure 4 depicts a controller between two tuples in the task simulation relation, in relation to the recipe and resources in Figures 2 and 1c.

We use function $\omega : \text{SIM} \times Lang(\mathcal{T}) \mapsto 2^{\underline{L}^+}$ in order to "read" the information stored in the controller's transitions. Given the recipe transition (task expression) that we want to execute, ω defines the set of possible corresponding sequences of transitions in the topology. Formally, $\omega(\theta, \mathcal{T}) = \{(\mathbf{t}_1 \cdots \mathbf{t}_k) \mid \exists \theta' : \theta' = \delta(\theta, \mathcal{T}, (\mathbf{t}_1 \cdots \mathbf{t}_k))\}$.

In Section 5, we defined what it means for a recipe to be realisable on a topology. Similarly, we define here what it means for a recipe to be *executed* on a topology by a controller. First, we say that a trace $\pi = s_0\xrightarrow{\mathcal{T}_1}s_1\xrightarrow{\mathcal{T}_2}\cdots\xrightarrow{\mathcal{T}_n}s_n$ of a recipe $\boldsymbol{R}$ is executed on a topology $\boldsymbol{P}$ by a controller $\boldsymbol{C}$ if and only if

1. it holds by backward induction that for each $i \in [0, n-1]$, we have $\omega((\underline{s}_i, s_i, \mathbf{r}_i), \mathcal{T}_{i+1}) \neq \emptyset$ with $(\underline{s}_i, s_i, \mathbf{r}_i) = \delta((\underline{s}_{i-1}, s_{i-1}, \mathbf{r}_{i-1}), \mathcal{T}_i, (\mathbf{t}_1 \cdots \mathbf{t}_k))$ for some $(\mathbf{t}_1 \cdots \mathbf{t}_k) \in \omega((\underline{s}_{i-1}, s_{i-1}, \mathbf{r}_{i-1}), \mathcal{T}_i)$;
2. $(\underline{s}_0, s_0, \mathbf{r}_0) = (\underline{s}^0, s^0, \mathbf{r}^0)$ is the initial state.

That is, we first use ω to return the set of transitions in the topology that we may choose in order to realise the current task expression $\mathcal{T}_i$ from the current state $(\underline{s}_i, s_i, \mathbf{r}_i)$ in the controller, and then use δ to compute the next state $(\underline{s}_{i+1}, s_{i+1}, \mathbf{r}_{i+1})$, which corresponds to the new state $\underline{s}_{i+1}$ in the topology, the new state s_{i+1} in the recipe (as it appears in π) and the new allocation $\mathbf{r}_{i+1}$. This iterative process is repeated at each step of π until the trace is completed, and can proceed (the set of choices is always non-empty) for any sequence returned by ω. In other words, a trace of the recipe is executed by a controller if, from their respective initial states, the controller can always associate the current task expression in the trace to at least one sequence of transitions in the topology realising that task, thereby orchestrating the resources such that the task expression is realised. A recipe $\boldsymbol{R}$ is executed on a topology $\boldsymbol{P}$ by a controller $\boldsymbol{C}$ iff all its traces are executed on $\boldsymbol{P}$ by $\boldsymbol{C}$.

Theorem 2 *Given $\boldsymbol{R}$ and $\boldsymbol{P}$ as above, any controller $\boldsymbol{C}$ (for $\boldsymbol{R}$ and $\boldsymbol{P}$) is such that $\boldsymbol{R}$ is executed on $\boldsymbol{P}$ by $\boldsymbol{C}$.*

PROOF. Let us proceed by induction on the length of traces π in $\boldsymbol{R}$. If $\pi = s_0$ then the claim holds: point 1 above does not apply because $0 = n$, and point 2 is trivially true. If $\pi = s_0 \xrightarrow{\mathcal{T}_1} s_1 \xrightarrow{\mathcal{T}_2} \cdots \xrightarrow{\mathcal{T}_{n-1}} s_{n-1}$ of $\boldsymbol{R}$ is executed by $\boldsymbol{C}$ on $\boldsymbol{P}$, assume that $\pi \xrightarrow{\mathcal{T}_n} s_n$ is not, i.e., $\omega((\underline{s}_{n-1}, s_{n-1}, \mathbf{r}_{n-1}), \mathcal{T}_n) = \emptyset$. By definition of ω, this means that there is no transition $(\underline{s}_n, s_n, \mathbf{r}_n) = \delta((\underline{s}_{n-1}, s_{n-1}, \mathbf{r}_{n-1}), \mathcal{T}_n, (\mathbf{t}_1 \cdots \mathbf{t}_k))$ for any $(\mathbf{t}_1 \cdots \mathbf{t}_k)$ and successor $(\underline{s}_n, s_n, \mathbf{r}_n)$. By definition of δ this also implies that either there is no $(\underline{s}_n, \mathbf{r}_n)$ in $\textsc{nxt}((\underline{s}_{n-1}, \mathbf{r}_{n-1})$ or no $(\underline{s}_n, s_n, \mathbf{r}_n) \in \textsc{transOf}((\underline{s}_{n-1}, \mathbf{r}_{n-1}), \mathcal{T}_n, (\underline{s}_n, \mathbf{r}_n))$. From either case it follows that $(\underline{s}_n, \mathbf{r}_n) \notin \textsc{nxt}^*((\underline{s}_0, \mathbf{r}_0), \pi)$, and thus π is not realisable. Then, by Theorem 1, no (non-empty) task simulation relation exists, and no controller can be defined. □

We conclude this section with a discussion about complexity.

Theorem 3 (Complexity) *Given $\boldsymbol{R}$ and $\boldsymbol{P}$, checking the existence of (and computing) a controller is polynomial on the size of the topology and exponential in the size of the recipe and number of resources.*

Intuitively, for any atomic expression in the recipe we need to consider any possible combination $(\underline{s}, \mathbf{r})$ of a new state in the topology together with any possible new resource vector (corresponding to recursive applications of the definition of $\textsc{nxt}$) to check whether there exists an allocation of the task (to a transition in the topology) such that $\mathbf{r}$ is the resulting vector. The size of the topology is exponential in the number of resources, and polynomial in their size. The size of all possible resource vectors is exponential in the number of resources and number of parts mentioned in the recipe. As the recipe can be equivalently represented as a rooted tree, if we take the size of the recipe as the maximum length of the sequences of atomic expressions (for any linearisation) appearing along its traces, then computing a simulation relation has cost $O((|\boldsymbol{P}| \times |V|)^{|\boldsymbol{R}|})$. We conclude by noting that we can compute the transitions in the controller (namely, the sequences of transitions in the topology that allow to realise each atomic task) *while* computing the simulation relation (see points (4.1) and (4.2) in Definition 4, and the discussion before Definition 6).

Our complexity result is consistent with that of [9], which is exponential in the number of available behaviours and polynomial in the size of the so-called enacted system behaviour [9].

7 ALGORITHM FOR REALISABILITY AND SYNTHESIS

We shall now provide an algorithm that computes a task simulation relation, as well as a controller, from a given topology and recipe. The algorithm clarifies how certain steps in our definitions could be implemented. In particular, we show how linearisations can be considered incrementally, without computing them all at the outset; how the closure of unobservable transitions are computed; how termination is ensured; and finally, how controller transitions are gathered.

The algorithms use some additional auxiliary notions, which we define first. Given a possibly atomic task expression $\mathcal{T} = \mathcal{T}_1; \mathcal{T}_2; \ldots; \mathcal{T}_n, n > 0$, we define its first element as $\textsc{fst}(\mathcal{T}) = \mathcal{T}_1$ (where $\mathcal{T}_1$ is either atomic or an interleaved composition), and the rest of its elements as $\textsc{rst}(\mathcal{T}) = \mathcal{T}_2; \ldots; \mathcal{T}_n$ if $n > 1$, and as $\textsc{rst}(\mathcal{T}) = \epsilon$ if $n = 1$. We also extend these notions as follows. Given an interleaved composition $\mathcal{T} = \mathcal{T}_1 | \ldots | \mathcal{T}_n, n > 1$, and an atomic task expression $\mathcal{T}_a$, we define the following:

$$\begin{aligned}
\overline{\textsc{fst}}(\mathcal{T}) &= \{\mathcal{T}_1 : \mathcal{T}_i = \mathcal{T}_1; \ldots; \mathcal{T}_m, \mathcal{T}_1 \text{ is atomic}\}, \\
\overline{\textsc{rst}}(\mathcal{T}, \mathcal{T}_a) &= \{\mathcal{T}_1 | \ldots | \mathcal{T}_{i-1} | \mathcal{T}_{i+1} | \ldots | \mathcal{T}_n : \mathcal{T}_i = \mathcal{T}_a\} \cup \\
&\quad\ \{\mathcal{T}_1 | \ldots | \mathcal{T}_{i-1} | \mathcal{T}' | \mathcal{T}_{i+1} | \ldots | \mathcal{T}_n : \mathcal{T}_i = \mathcal{T}_a; \mathcal{T}'\}, \\
\textsc{fst}\mathrm{R}\textsc{st}(\mathcal{T}) &= \{\mathcal{T}_1; \mathcal{T}_2 : \mathcal{T}_1 \in \overline{\textsc{fst}}(\mathcal{T}), \mathcal{T}_2 \in \overline{\textsc{rst}}(\mathcal{T}, \mathcal{T}_1)\},
\end{aligned}$$

where $\mathcal{T}'$ is any non-empty sequence. These notions define the first elements of interleaved composition $\mathcal{T}$ as the set of all first elements in each of its sequences $\mathcal{T}_i$, and the rest of $\mathcal{T}$ (relative to some first element $\mathcal{T}_a$) as the set of all interleaved compositions obtained from $\mathcal{T}$ by removing $\mathcal{T}_a$ (once) if it occurs first in some $\mathcal{T}_i$.

Given a topology and recipe, Algorithms 1 and 2 work as follows. Algorithm 1 additionally takes a state s in the recipe and a couple $\sigma = (\underline{s}, \mathbf{r})$ as input, where $\underline{s}$ is a state in the topology and $\mathbf{r}$ is a resource vector. Then, for each outgoing transition in the recipe, from s to some s', the algorithm checks whether the associated task expression $\mathcal{T}$ can be simulated from the corresponding topology state $\underline{s}$, and (recursively) whether the same holds for each outgoing transition from s'. Thus, s' is passed as a parameter to Algorithm 2 in order to "remember" to continue from s'. Whenever the pair s and $\underline{s}$ are indeed in simulation, they are added to the set $\textsc{sim}$ in line 6, together with the corresponding resource vector $\mathbf{r}$. Observe that this is done in a post-order (depth-first) fashion, from the "deepest" states in the recipe to the state that was passed into Algorithm 1 when it was first called. For controller synthesis, the algorithm also passes the entire recipe transition being considered and its corresponding couple σ as respectively the third and fifth parameters (line 3) to Algorithm 2.

Intuitively, Algorithm 2 implements the $\textsc{nxt}$ operator in Definition 4. Basically, the algorithm performs a depth-first traversal of the topology to check whether task expression $\mathcal{T}_{cur}$ (initially $\mathcal{T}$) can be simulated. We highlight the important lines below. Line 6 looks for a transition in the topology from state $\underline{s}_{nxt}$ that matches the first element in $\mathcal{T}_{cur}$, provided that the couple σ_{nxt} was not already explored in a previous (recursive) step. Lines 11 and 12 incrementally check each linearisation of an interleaved composition $\mathcal{T}_{cur}$, by recursively checking the concatenation of each first element in $\mathcal{T}_{cur}$, with a corresponding remainder. The loop exits as soon as any viable linearisation is found. Line 14 is only applicable if $\mathcal{T}_{cur}$ could not be simulated so far, and the topology-transition being considered is an unobservable one. In this case, the transition is "skipped", and a simulation of $\mathcal{T}_{cur}$ is tried from the next state $\underline{s}'$ (with its corresponding resource allocation). Line 2 is only applicable when $\mathcal{T}_{cur}$ has been completely "processed"; it is at this point that a transition x is added to the controller's transition function δ (Definition 6). Finally, line 16 keeps track of the fact that σ_{nxt} is being explored, in order to guarantee termination. Combined with the fact that the recipe is acyclic, the following results hold trivially.

Proposition 4 *Algorithm 1 always terminates. Furthermore, it is optimal with respect to computational complexity.*

Observe that line 16 (and other lines) in Algorithm 2 adds the current transition label $\mathbf{t}$ to the end of the sequence of topology

```
Algorithm 1 FindSim($\sigma, s$)
Input: Topology $(\underline{s}^0, \underline{S}, \underline{L}, \Longrightarrow)$ and recipe $(s^0, S, L, \rightarrow)$;
       the couple $\sigma = (\underline{s}, \mathbf{r})$; state $s \in S$.
Output: Either $(\emptyset, \emptyset)$ or a non-empty relation $\textsc{sim}$ and function $\delta$.
1: $\delta, \textsc{sim}, \textsc{sim}' := \emptyset$
2: for each $(s, \mathcal{T}, s') \in \rightarrow$ do
3:     $(\textsc{sim}', \delta) :=$ EvalExp$(\sigma, \mathcal{T}, (s, \mathcal{T}, s'), \epsilon, \sigma, \emptyset)$
4:     if $\textsc{sim}' = \emptyset$ then return $(\emptyset, \emptyset)$
5:     $\textsc{sim} := \textsc{sim} \cup \textsc{sim}'$
6: return $(\textsc{sim} \cup \{(\underline{s}, s, \mathbf{r})\}, \delta)$
```

```
Algorithm 2 EvalExp($\sigma_{nxt}, \mathcal{T}_{cur}, tr, (\mathbf{t}_1 \cdots \mathbf{t}_k), \sigma_0, \Sigma$)
Input: Topology $(\underline{s}^0, \underline{S}, \underline{L}, \Longrightarrow)$ and recipe $(s^0, S, L, \rightarrow)$;
       current successor couple $\sigma_{nxt} = (\underline{s}_{nxt}, \mathbf{r}_{nxt})$;
       task expression being processed $\mathcal{T}_{cur}$;
       current recipe-transition $tr = (s, \mathcal{T}, s_{nxt})$;
       current sequence of topology-transition labels $(\mathbf{t}_1 \cdots \mathbf{t}_k)$;
       couple $\sigma_0 = (\underline{s}_0, \mathbf{r}_0)$, corresponding to $s$; visited couples $\Sigma$.
Output: Either $(\emptyset, \emptyset)$ or a non-empty relation $\textsc{sim}$ and function $\delta$.
1: $\delta, \textsc{sim} := \emptyset$
2: if $\mathcal{T}_{cur} = \epsilon$ then
3:     $x := (\underline{s}_0, s, \mathbf{r}_0) \xrightarrow{\mathcal{T}, (\mathbf{t}_1 \cdots \mathbf{t}_k)} (\underline{s}_{nxt}, s_{nxt}, \mathbf{r}_{nxt})$
4:     $(\textsc{sim}, \delta) :=$ FindSim$(\sigma_{nxt}, s_{nxt})$
5:     return $(\textsc{sim}, \delta \cup \{x\})$
6: for each $(\underline{s}_{nxt}, \mathbf{t}, \underline{s}') \in \Longrightarrow$ with $\sigma_{nxt} \notin \Sigma$ do
7:     if $\textsc{fst}(\mathcal{T}_{cur}) = t_1 \| \ldots \| t_n$ and
           $\mathbf{r}'' \in \textsc{al}(\mathbf{r}_{nxt}, \textsc{fst}(\mathcal{T}_{cur}), \mathbf{t})$ then
8:         $\sigma' := (\underline{s}', \mathbf{r}')$, where $\mathbf{r}' = \textsc{mov}(\mathbf{r}'', \mathbf{t})$
9:         $(\textsc{sim}, \delta) :=$
               EvalExp$(\sigma', \textsc{rst}(\mathcal{T}_{cur}), tr, (\mathbf{t}_1 \cdots \mathbf{t}_k \cdot \mathbf{t}), \sigma_0, \emptyset)$
10:    else if $\textsc{fst}(\mathcal{T}_{cur}) = \mathcal{T}_1 | \ldots | \mathcal{T}_n$ (where $n > 1$) then
11:        for each $\mathcal{T}' \in \textsc{fst}\textsc{Rst}(\mathcal{T}_{cur})$ do
12:            $(\textsc{sim}, \delta) :=$ EvalExp$(\sigma_{nxt}, \mathcal{T}', tr, (\mathbf{t}_1 \cdots \mathbf{t}_k), \sigma_0, \emptyset)$
13:            if $\textsc{sim} \neq \emptyset$ then return $(\textsc{sim}, \delta)$
14:    if $\textsc{sim} = \emptyset$ and $\forall t \in \mathbf{t}, t \notin T_{ob}$ then
15:        $\sigma' := (\underline{s}', \mathbf{r}')$, where $\mathbf{r}' = \textsc{mov}(\mathbf{r}_{nxt}, \mathbf{t})$
16:        $(\textsc{sim}, \delta) :=$
               EvalExp$(\sigma', \mathcal{T}_{cur}, tr, (\mathbf{t}_1 \cdots \mathbf{t}_k \cdot \mathbf{t}), \sigma_0, \Sigma \cup \{\sigma_{nxt}\})$
17:    if $\textsc{sim} \neq \emptyset$ then return $(\textsc{sim}, \delta)$
18: return $(\emptyset, \emptyset)$
```

transitions $\mathbf{t}_1 \cdots \mathbf{t}_k$ being pursued. This essentially implements the TRANSOF operator described in Section 6, and together with line 3, leads to the fact that the algorithm correctly computes a controller for the recipe and topology. Moreover, the algorithm also correctly computes a task simulation relation for them.

Theorem 5 (Correctness) *Let* $\boldsymbol{P} = (\underline{s}^0, \underline{S}, \underline{L}, \Longrightarrow)$ *be a topology and* $\boldsymbol{R} = (s^0, S, L, \rightarrow)$ *a recipe. Let* $(\textsc{sim}, \delta) =$ *FindSim*(σ, s^0) *with* $\sigma = (\underline{s}^0, \mathbf{r}^0)$. *Then* (a) $\textsc{sim} = \emptyset$ *iff* $\boldsymbol{R}$ *is not realisable in* $\boldsymbol{P}$*; otherwise,* (b) $\textsc{sim}$ *is a task simulation relation between* $\boldsymbol{P}$ *and* $\boldsymbol{R}$*; and* (c) $(\textsc{sim}, \delta)$ *is a controller of* $\boldsymbol{P}$ *and* $\boldsymbol{R}$.

PROOF SKETCH. This is an involved proof by induction on the "height" h of vertices in the recipe. The main part involves showing that SIM is a task simulation relation between $\boldsymbol{P}$ and $\boldsymbol{R}$. For the base case, we take $h = 0$. Then, there is no recipe transition $(s, \mathcal{T}, \underline{s}') \in \rightarrow$ for any $\mathcal{T}$ (line 2 in Algorithm 1). Thus, the algorithm returns $\textsc{sim} = \{(\underline{s}, s, \mathbf{r})\}$, and $(\underline{s}, s, \mathbf{r}) \in \textsc{sim}$ also holds by Definition 5. Next, we assume that the theorem holds if $h \geq x$, for some $x \in \mathbb{N}_0$ (induction hypothesis). For the inductive step, we take $h = x + 1$. Then, there must exist a recipe transition $(s, \mathcal{T}, \underline{s}') \in \rightarrow$ for some $\mathcal{T}$. We prove that if $\textsc{sim}' \neq \emptyset$ in line 3, then $\textsc{sim}'$ is "sound". We do this by induction on the length of expression $\mathcal{T}_{cur}$ in Algorithm 2. The main point is that the couples in Σ' and Σ in steps (4.1) and (4.2) in Definition 4 are the same as in lines 8 and 15 in Algorithm 2, respectively. Finally, if $\textsc{sim}' \neq \emptyset$ in line 3 of Algorithm 1 for all transitions considered in line 2, it follows that $(\underline{s}, s, \mathbf{r}) \in \textsc{sim}$ also holds by the induction hypothesis and Definition 5. □

8 DISCUSSION AND FUTURE WORK

In this paper we presented an approach for modelling a flexible manufacturing system and checking whether it is possible to manufacture a given product in the system, represented as a manufacturing recipe. This check is done by computing a task simulation relation as discussed in Section 5. Further, we presented an approach for synthesising *a* controller that is able to orchestrate the manufacturing resources in the system so as to realise the recipe.

However, we did not address the problem of synthesising *any* controller for the given recipe. Rather, *(a)* we computed *a possible* task simulation relation, and *(b)* defined *a possible* controller. Moreover, the algorithm presented in Section 7 returns a controller which represents *exactly one* way of orchestrating the resources to realise the recipe, as this is sufficient for our application.

One direction for future research therefore involves the synthesis of *any* controller for a given recipe, and the technical content of this paper already allows for this extension. Computing any possible controller amounts to *(a)* computing the *largest* task simulation relation, and *(b)* including in the controller, between two states $(\underline{s}, s, \mathbf{r})$ and $(\underline{s}', s', \mathbf{r}')$, all possible sequences of transitions defined by $\textsc{transOf}((\underline{s}, \mathbf{r}), \mathcal{T}, (\underline{s}', \mathbf{r}'))$ and not just at least one, as in Definition 6. An algorithm can be devised accordingly.

Another research direction involves checking realisability (and synthesising suitable controllers) *offline*, but for *classes* of recipes instead of single recipes. A class of recipes can be expressed in terms of a union of recipes, but also as unbounded (even cyclic) processes, which is already handled by our notion of task simulation, as it relies on the standard mathematical notion of simulation. To check the realisability of a new recipe, it would therefore be sufficient to check whether it can be simulated (task simulation) by one such class, which is a polynomial check in the size of the recipe.

Finally, although our current formalism allows only parallel tasks on *different* parts, we plan to extend it to capture parallel tasks operating on *the same* part (e.g. two machines performing a joint task on a single part). Allowing parallel tasks on the same part requires machines to be able to "see" the workspaces of each other, thus increasing the complexity of the algorithm proportionately.

ACKNOWLEDGEMENTS

This research was funded by EPSRC grants EP/K018205/1 and EP/K014161/1, the support of which is gratefully acknowledged. We would like to thank Natasha Alechina for many helpful ideas and discussions relating to the work presented here, and Nikolas Antzoulatos and Elkin Castro for useful advice and material.

REFERENCES

[1] ANSI/ISA-95, Enterprise-Control System Integration, Parts 1-5.
[2] Business To Manufacturing Markup Language - Operations Schedule - Version 6.0.
[3] IEC 62264-1: Enterprise-control system integration - Parts 1-3.
[4] 'A Landscape for the Future of High Value Manufacturing in the UK', Technical report, Technology Strategy Board, (2012).

[5] Zhuming M. Bi, Sherman Y.T. Lang, Weiming Shen, and Lihui Wang, 'Reconfigurable manufacturing systems: the state of the art', *International Journal of Production Research*, **46**(4), 967–992, (2008).

[6] Simon Bliudze and Joseph Sifakis, 'The algebra of connectors: Structuring interaction in BIP', in *Proceedings of the ACM IEEE International Conference on Embedded Software*, EMSOFT '07, pp. 11–20, (2007).

[7] Jim Browne, Didier Dubois, Keith Rathmill, Suresh P. Sethi, and Kathryn E. Stecke, 'Classification of flexible manufacturing systems', *The FMS magazine*, **2**(2), 114–117, (1984).

[8] J.C. Chaplin, O.J. Bakker, L. de Silva, D. Sanderson, E. Kelly, B. Logan, and S.M. Ratchev, 'Evolvable assembly systems: A distributed architecture for intelligent manufacturing', *IFAC-PapersOnLine*, **48**(3), 2065–2070, (2015).

[9] Giuseppe De Giacomo, Fabio Patrizi, and Sebastian Sardina, 'Automatic behavior composition synthesis', *AIJ*, **196**, 106–142, (2013).

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

[11] Kutluhan Erol, James Hendler, and Dana S. Nau, 'HTN planning: Complexity and expressivity', in *Proceedings of the National Conference on Artificial Intelligence (AAAI-94)*.

[12] 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*, Forschungsunion, 2013.

[13] Yoram Koren, Uwe Heisel, Francesco Jovane, Toshimichi Moriwaki, G. Pritschow, G. Ulsoy, and H. Van Brussel, 'Reconfigurable manufacturing systems', *CIRP Annals-Manufacturing Technology*, **48**(2), 527–540, (1999).

[14] Yuqian Lu, Xun Xu, and Jenny Xu, 'Development of a hybrid manufacturing cloud', *Journal of Manufacturing Systems*, **33**(4), 551–566, (2014).

[15] Mostafa G. Mehrabi, A. Galip Ulsoy, and Yoram Koren, 'Reconfigurable manufacturing systems: key to future manufacturing', *Journal of Intelligent Manufacturing*, **11**(4), 403–419, (2000).

[16] Robin Milner, 'An algebraic definition of simulation between programs', Technical report, Stanford University, (1971).

[17] Tadao Murata, 'Petri nets: Properties, analysis and applications', *Proceedings of the IEEE*, **77**(4), 541–580, (1989).

[18] 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*, pp. 42–55, (1996).

[19] Chris Rhodes, *Manufacturing: Statistics and Policy. Briefing Paper*, House of Commons Library, 2015.

[20] Andrea Krasa Sethi and Suresh Pal Sethi, 'Flexibility in manufacturing: a survey', *International Journal of Flexible Manufacturing Systems*, **2**(4), 289–328, (1990).

[21] Daniel Smale and Svetan Ratchev, 'A capability model and taxonomy for multiple assembly system reconfigurations', in *Proceedings of IFAC Symposium on Information Control Problems in Manufacturing (INCOM)*, volume 13, pp. 1923–1928, (2009).

[22] W. Murray Wonham and Peter J. Ramadge, 'On the supremal controllable sub-language of a given language', *SIAMJCO*, **25**(3), 637–659, (1987).