Compositional Specification of Functionality and Timing of Manufacturing Systems

Bram van der Sanden*, João Bastos*, Jeroen Voeten*‡, Marc Geilen*, Michel Reniers*,
Twan Basten*‡, Johan Jacobs†, and Ramon Schiffelers*†
*Eindhoven University of Technology, Eindhoven, The Netherlands
b.v.d.sanden@tue.nl
†ASML, Veldhoven, The Netherlands
‡TNO Embedded Systems Innovation, Eindhoven, The Netherlands

***Abstract*—This paper introduces a formal modeling approach for compositional specification of both functionality and timing of manufacturing systems. Functionality aspects can be considered orthogonally to timing aspects. The functional aspects are specified using two abstraction levels; high-level activities and lower level actions. Design of a functionally correct controller is possible by looking only at the activity level, abstracting from the different execution orders of actions and their timing. As a result, controller design can be performed on a much smaller state space compared to an explicit model where timing and actions are present. The performance of the controller can be analyzed and optimized by taking into account the timing characteristics. Since formal semantics are given in terms of a $(\max, +)$ state space, various existing performance analysis techniques can be used. We illustrate the approach, including performance analysis, on an example manufacturing system.**

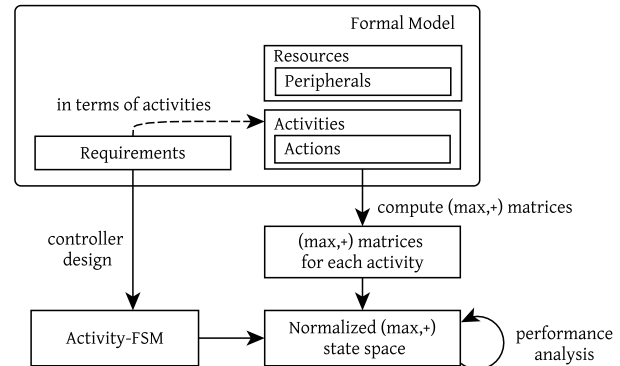


Fig. 1. Overview of the modeling approach.

I. Introduction

One of the challenges in the design of manufacturing systems is the development of supervisory control components. Due to increasing complexity of these systems, design of these components is becoming more difficult. In such systems, supervisory controllers play a role to guarantee functional correctness, for instance, to prevent unsafe behavior of the system such as product or robot collisions in a shared physical area. Besides functional correctness, the controller must also optimize performance criteria such as maximizing throughput or minimizing makespan. In order to perform this optimization, the timing characteristics of the system are necessary.

In this paper, we introduce a formal modeling approach, shown in Fig. 1, to address both functionality and timing aspects of manufacturing systems in a compositional way. System operations are modeled as so called *activities*. Activities are specified as directed acyclic graphs, which consist of (1) a set of actions executed on resources, and (2) a set of dependencies among the actions. Functional requirements related to activity sequences are modularly and concisely specified using automata. These requirements can for instance enforce product life cycles and ensure safety [1]. The composition of these automata is characterized using multiparty synchronization, where execution of shared events among different automata is synchronous. The advantage of multiparty synchronization is that requirements can be added in a modular way, and are respected after composition.

Controller design is performed on the activity level, abstracting from the internals (actions) of activities. This means that at the controller level, there is no redundant interleaving from different execution orders of fine-grained actions. Furthermore, we abstract from the specific timing of actions. As a result, the state space of the controller is much smaller compared to an explicit model where timing and actions are present.

Performance analysis and optimization of the supervisory controller requires the dynamic semantics (the timing behavior) of our modeling approach. The dynamic semantics of activities is expressed using matrices in $(\max, +)$ linear algebra (see for instance [2]). These matrices abstract from the internal graph structure, which has again an advantage in terms of scalability. Activity sequences are captured by $(\max, +)$ automata [3], which can also be represented by a $(\max, +)$ state space. These automata combine matrices with nondeterministic choices, corresponding to choices in the ordering of activities. Finding a throughput-optimal controller corresponds to finding an optimal repeatable activity sequence in the state space, which can be found using existing optimal cycle ratio algorithms [4].

The modeling approach presented in this paper can be taken as a semantic underpinning, on top of which a domain specific language (DSL) is put, allowing system engineers to model a complete system. Our approach is already in use within ASML[1], the world-leading manufacturer of lithography

[1] www.asml.com

systems, to formalize the specification of the product handling part of their machines. The DSL that is put on top describes the system in terms of resources, actions, symbolic positions of motors, and activities that can be performed.

The remainder of the paper is structured as follows. Section II describes the modeling concepts, and the static and dynamic semantics of both activities and activity sequences. Section III illustrates the use of the modeling approach by an example manufacturing system. Both the activities and the allowed activity sequences are modeled concisely. The model is used to find a throughput-optimal controller for the system. Section IV describes how the modeling formalism is being used in industry. Related work is given in Section V, and Section VI concludes the paper and describes future extensions that are currently being investigated.

II. Modeling Concepts

In this section, we introduce the formal semantics of our modeling approach. First we focus on describing all possible machine behaviors. Then, we describe activities and activity sequences that allows to decribe useful behavior.

We view the system as consisting of a set of *peripherals*. Each of these peripherals can execute *actions*. The complete set of actions describes all behavior that the machine can exhibit. Peripherals are aggregated into *resources*, which can be *claimed* and *released*. As an example, consider a robot resource that can move products. This robot has a number of peripherals that can perform actions, such as a clamp that can hold or release a product, or a robot motor to move the robot.

In manufacturing systems there are often operations of coordinated actions, that describe a scenario of deterministic behavior. For instance, picking up a product and placing it on another processing station, or performing a fixed operation on the product. These entire operations are modeled as single *activities*, consisting of a fixed set of actions and dependencies among them. A supervisory controller influences the order in which activities can be executed, but not the order of actions in an activity. Fig. 2 gives a schematic overview of the modeling concepts and the different layers.

In the remainder of this section, we describe the static and dynamic semantics of both activities and activity sequences.

A. Static Semantics

The following sets define the basic elements of our model:

- set $\mathcal{A}$ of actions, with typical elements $a \in \mathcal{A}$;
- set $\mathcal{P}$ of peripherals, with typical elements $p \in \mathcal{P}$;
- set $\mathcal{R}$ of resources, with typical elements $r \in \mathcal{R}$.

We assume a function $R : \mathcal{P} \to \mathcal{R}$, such that $R(p)$ is the resource that contains p.

Given the basic elements of the language, we now introduce the notion of an activity, and define its structure. As a running example, we have the three activities shown in Fig. 3. Activities are directed acyclic graphs (DAGs), consisting of a set of actions executed on resources, and dependencies among those actions. Nodes refer to either an action executed by a peripheral, or a claim or release of a resource. In Fig. 3, nodes are annotated with their mapping, and the value in the node is the execution time. The colors indicate peripheral actions included in a certain activity.

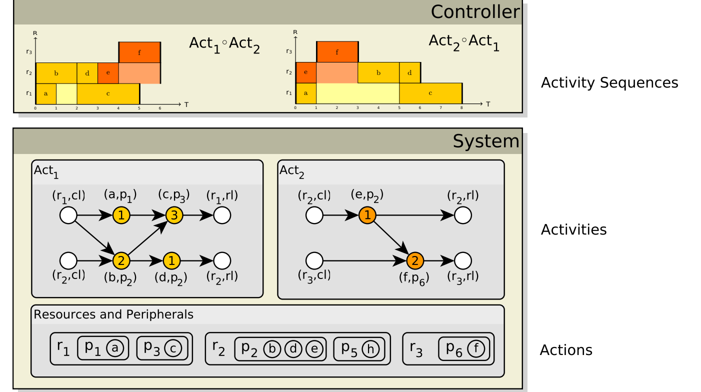


Fig. 2. Schematic overview of the concepts in our formal modeling approach. The system is modeled using resources consisting of a number of peripherals, which can execute actions. Activities describe deterministic operations in the system. A supervisory controller controls the system by influencing the order of activity execution.

Definition 1. *An activity is a DAG $(N, \to)$, consisting of a set N of nodes and a set $\to \subseteq N \times N$ of dependencies. We write a dependency $(a, b) \in \to$ as $a \to b$, We assume a mapping function $M : N \to \mathcal{A} \times \mathcal{P} \cup \mathcal{R} \times \{rl, cl\}$, which associates a node to either a pair (a, p) referring to an action executed on a peripheral; or to a pair (r, v) with $v \in \{rl, cl\}$, referring to a* claim *(cl) or* release *(rl) of resource r. Nodes mapped to a pair (a, p) are called* action nodes*, and nodes mapped to a claim or release of a resource are called* claim *and* release *nodes respectively.*

We assume a number of constraints that ensure that activities can be statically checked for proper resource claiming. These constraints are however not strictly necessary for timing analysis.

- All nodes mapped to the same peripheral are sequentially ordered to avoid self-concurrency;
- Each resource is claimed no more than once;
- Each resource is released no more than once;
- Every action node is preceded by a claim node on the corresponding resource;
- Every action node is succeeded by a release node on the corresponding resource;
- Every release node is preceded by a claim node on the corresponding resource;
- Every claim node is succeeded by a release node on the corresponding resource.

For each activity, we define the set of resources it uses, which is needed in the later definition of sequencing activities.

Definition 2 (Resources of Activity)**.** *Given activity $Act = (N, \to)$, we define set*
$R(Act) = \{r \in \mathcal{R} \mid (\exists n \in N \mid M(n) = (r, cl))\}$.

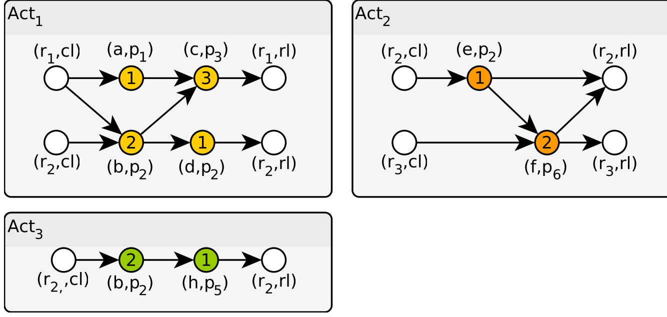


Fig. 3. Activities Act_1, Act_2, and Act_3.

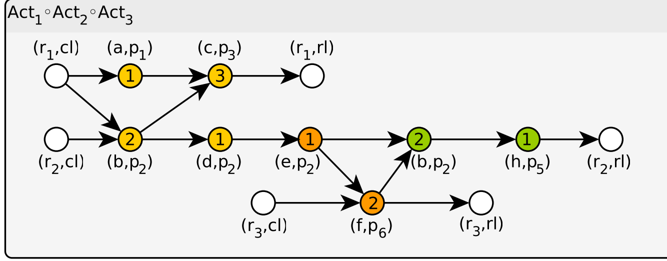


Fig. 4. Activity $Act_1 \cdot Act_2 \cdot Act_3$.

Multiple activities can be composed into a combined activity using the sequencing operator. Given the set of shared resources, it removes intermediate release and claim nodes on these resources, and properly links the dependencies. This form of sequencing is similar to the notion of weak sequential composition [5], which is also defined relative to a dependency relation over a set of actions.

Definition 3 (Sequencing Operator)**.** *Given two activities $Act_1 = (N_1, \rightarrow_1)$ and $Act_2 = (N_2, \rightarrow_2)$ with $N_1 \cap N_2 = \emptyset$, we define $Act_1 \cdot Act_2$ as activity $Act_{1\cdot 2} = (N_{1\cdot 2}, \rightarrow_{1\cdot 2})$.*

Let $R_{1\cap 2} = R(Act_1) \cap R(Act_2)$ denote the set of resources used in both activities. Define the set of corresponding release nodes in N_1, and claim nodes in N_2 as $rl_{1\cap 2} = \{n_1 \mid n_1 \in N_1 \wedge (\exists r \in R_{1\cap 2} \mid M(n_1) = (r, rl))\}$, and $cl_{1\cap 2} = \{n_2 \mid n_2 \in N_2 \wedge (\exists r \in R_{1\cap 2} \mid M(n_2) = (r, cl))\}$ respectively.

Activity $Act_{1\cdot 2} = (N_{1\cdot 2}, \rightarrow_{1\cdot 2})$ is now defined as follows:

$$\begin{aligned} N_{1\cdot 2} &= (N_1 \cup N_2) \backslash (cl_{1\cap 2} \cup rl_{1\cap 2}) \\ \rightarrow_{1\cdot 2} &= \{(n_i, n_j) \mid \ n_i \rightarrow_1 n_j \wedge \ n_j \notin rl_{1\cap 2}\} \cup \\ &\quad \{(n_i, n_j) \mid \ n_i \rightarrow_2 n_j \wedge \ n_i \notin cl_{1\cap 2}\} \cup \\ &\quad \{(n_1, n_2) \mid \ (\exists n_{rl} \in rl_{1\cap 2} \mid n_1 \rightarrow_1 n_{rl}) \wedge \\ &\qquad (\exists n_{cl} \in cl_{1\cap 2} \mid n_{cl} \rightarrow_2 n_2)\}. \end{aligned}$$

Fig. 4 shows how activities Act_1, Act_2, and Act_3, shown in Fig. 3, are composed to activity $Act_1 \cdot Act_2 \cdot Act_3$ using the sequencing operator. Note that the sequencing operator is associative.

B. Dynamic Semantics

So far, we have described the structure of activities and the way they can be composed. In order to do performance analysis, we need to introduce timing information. We do so on action level, activity level, and activity sequence level.

Definition 4 (Execution time of an action)**.** *We assume a function $T : \mathcal{A} \rightarrow \mathbb{R}_{\geq 0}$ that maps each action to its fixed execution time.*

Definition 5 (Execution time of a node)**.** *We define a function $T : N \rightarrow \mathbb{R}_{\geq 0}$ that maps each node to a fixed execution time, given a node $n \in N$ in activity $(N, \rightarrow)$:*

$$T(n) = \begin{cases} T(a) & \text{if } M(n) = (a, p) \\ & \text{for some } a \in \mathcal{A}, p \in \mathcal{P} \\ 0 & \text{otherwise.} \end{cases}$$

We use $(\max, +)$ algebra to capture the dynamic semantics of activities in a concise way. Two essential characteristics of the execution of an activity are *synchronization*; when a node waits for all its incoming dependencies to finish, and *delay*; when an action execution starts, it takes a fixed amount of time before it completes. These characteristics correspond well to the $(\max, +)$ operators *max* and *addition*, defined over the set $\mathbb{R}^{-\infty} = \mathbb{R} \cup \{-\infty\}$. The $\max$ and $+$ operators are defined as in usual algebra, with the additional convention that $-\infty$ is the unit element of $\max$: $\max(-\infty, x) = \max(x, -\infty) = x$, and the zero-element of addition: $-\infty + x = x + -\infty = -\infty$. Addition distributes over the maximum operator: $x + \max(y, z) = \max(x + y, x + z)$.

To formalize synchronization we need a notion of predecessor nodes.

Definition 6 (Predecessor nodes)**.** *Given activity $(N, \rightarrow)$ and node $n \in N$, we define the set of predecessor nodes:*

$$Pred(n) = \{n_{in} \in N \mid n_{in} \rightarrow n\}.$$

Since actions are executed on resources, we assume a *resource time stamp* vector $\gamma_R : \mathcal{R} \rightarrow \mathbb{R}^{-\infty}$. The vector represents the system state in terms of resource availability. Each entry $\gamma_R(r)$ corresponds to the availability time of the resource r in the system. These entries are used to determine when resources are available, and hence can be claimed. All entries in the initial vector are assumed to be zero, to indicate that all resources are available upon start of the system.

Definition 7 (Start and completion time of a node)**.** *Given activity $Act = (N, \rightarrow)$ and resource time stamp vector γ_R, we can define the start time $start(n)$ and completion time $end(n)$ for each node $n \in N$:*

$$start(n) = \begin{cases} \gamma_R(r) & \text{if } M(n) = (r, cl) \\ \max\limits_{n_{in} \in Pred(n)} end(n_{in}) & \text{otherwise} \end{cases}$$

$$end(n) = start(n) + T(n).$$

Action a can start as soon as all predecessor actions completed execution. Note that the start and end times for each

node are uniquely defined, due to the structural properties of activities. This also means that the dynamic semantics of an activity $Act = (N, \rightarrow)$ is uniquely defined by N, $\rightarrow$, and a timing function T,

Now, consider a resource time stamp vector γ_R as starting configuration of the system. After execution of activity $Act = (N, \rightarrow)$, we get a new resource time stamp vector γ'_R, where each entry is defined as follows:

$$\gamma'_R(r) = \begin{cases} \gamma_R(r) & \text{if } r \notin R(Act) \\ end(n) & \text{if } r \in R(Act) \wedge M(n) = (r, rl) \\ & \quad \text{for some } n \in N. \end{cases}$$

Since $(\max, +)$ algebra is a linear algebra, it can be extended to matrices and vectors in the usual way. Given matrix $\boldsymbol{A}$ and vector $\boldsymbol{x}$, we use $\boldsymbol{A} \otimes \boldsymbol{x}$ to denote the $(\max, +)$ matrix multiplication. Given $m \times p$ matrix $\boldsymbol{A}$ and $p \times n$ matrix $\boldsymbol{B}$, the elements of the resulting matrix $\boldsymbol{A} \otimes \boldsymbol{B}$ are determined by: $[A \otimes B]_{ij} = \max_{k=1}^{p}([A]_{ik} + [B]_{kj})$. For any vector $\boldsymbol{x}$, $\|\boldsymbol{x}\| = \max_i x_i$ denotes the vector norm of $\boldsymbol{x}$. For vector $\boldsymbol{x}$, with $\|\boldsymbol{x}\| > -\infty$, we use $\boldsymbol{x^{norm}}$ to denote $\boldsymbol{x} - \|\boldsymbol{x}\|$, the normalized vector, such that $\|\boldsymbol{x^{norm}}\| = 0$. We use $\boldsymbol{0}$ to denote a vector with all zero-valued entries.

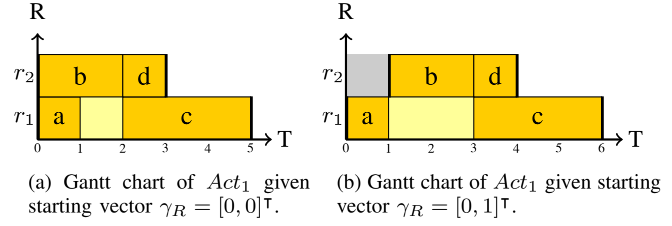
Using this linear algebra, we can capture the behavior of an activity in a $(\max, +)$ matrix. Consider activity Act, characterized by a $(\max, +)$ matrix $\boldsymbol{M}_{Act}$. Then, given a resource time stamp vector γ_R, the new vector γ'_R is given by $\gamma'_R = \boldsymbol{M}_{Act} \otimes \gamma_R$. An algorithm for computing the activity matrices automatically can be found in [6, Algorithm 1].

Example 8 ($(\max, +)$ characterization)**.** *Consider activity Act_1, shown in Fig. 3, with $T(a) = 1, T(b) = 2, T(c) = 3$ and $T(d) = 1$. Where $\mathcal{R} = \{r_1, r_2\}$, $R(p_1) = R(p_3) = r_1$, and $R(p_2) = r_2$. We start with a resource time stamp vector $\gamma_R = [r_1, r_2]^\intercal$. Now, the $(\max, +)$ expressions related to the ending time of the nodes are as follows:*

$$\begin{aligned}
end(cl(r_1)) &= \gamma_R(r_1) \\
end(cl(r_2)) &= \gamma_R(r_2) \\
end(a) &= \max(end(cl(r_1))) + T(a) = \gamma_R(r_1) + 1 \\
end(b) &= \max(end(cl(r_2))) + T(b) = \gamma_R(r_2) + 2 \\
end(c) &= \max(end(a), end(b)) + T(c) \\
&= \max(\gamma_R(r_1) + 1, \gamma_R(r_2) + 2) + 3 \\
&= \max(\gamma_R(r_1) + 4, \gamma_R(r_2) + 5) \\
end(d) &= \max(end(b)) + T(d) = \gamma_R(r_2) + 3 \\
end(rl(r_1)) &= end(c) = \max(\gamma_R(r_1) + 4, \gamma_R(r_2) + 5) \\
end(rl(r_2)) &= end(d) = \gamma_R(r_2) + 3.
\end{aligned}$$

The $(\max, +)$ characterization of $end(rl(r_j))$ for any r_j can be written in the normal form $r_j = \max_{r_i \in \mathcal{R}}(\gamma_R(r_i) + t_i)$ for some $t_i \in \mathbb{R}^{-\infty}$. Note that $t_i = -\infty$ for any $r_i \in \mathcal{R} \setminus R(Act_1)$. Written in normal form, we get:

$$\begin{aligned}
end(rl(r_1)) &= \max(\gamma_R(r_1) + 4, \gamma_R(r_2) + 5) \\
end(rl(r_2)) &= \max(\gamma_R(r_1) + -\infty, \gamma_R(r_2) + 3).
\end{aligned}$$



(a) Gantt chart of Act_1 given starting vector $\gamma_R = [0, 0]^\intercal$. (b) Gantt chart of Act_1 given starting vector $\gamma_R = [0, 1]^\intercal$.

Fig. 5. Gantt charts for Act_1 given different starting resource availability vectors.

The $(\max, +)$ characterization of the activity is

$$\boldsymbol{M}_{Act_1} = \begin{bmatrix} 4 & 5 \\ -\infty & 3 \end{bmatrix}.$$

Given γ_R, the new vector γ'_R is computed as follows:

$$\begin{aligned}
\boldsymbol{M}_{Act_1} \otimes \gamma_R &= \begin{bmatrix} 4 & 5 \\ -\infty & 3 \end{bmatrix} \otimes \begin{bmatrix} \gamma_R(r_1) \\ \gamma_R(r_2) \end{bmatrix} \\
&= \begin{bmatrix} \max(4 + \gamma_R(r_1), 5 + \gamma_R(r_2)) \\ \max(-\infty + \gamma_R(r_1), 3 + \gamma_R(r_2)) \end{bmatrix} \\
&= \begin{bmatrix} \gamma'_R(r_1) \\ \gamma'_R(r_2) \end{bmatrix}.
\end{aligned}$$

For example, given starting vector $\gamma_R = [0 \ 1]^\intercal$, γ'_R is computed as:

$$\begin{aligned}
\boldsymbol{M}_{A_1} \otimes \gamma_R &= \begin{bmatrix} 4 & 5 \\ -\infty & 3 \end{bmatrix} \otimes \begin{bmatrix} 0 \\ 1 \end{bmatrix} \\
&= \begin{bmatrix} \max(4 + 0, 5 + 1) \\ \max(-\infty + 0, 3 + 1) \end{bmatrix} = \begin{bmatrix} 6 \\ 4 \end{bmatrix}.
\end{aligned}$$

Fig. 5 shows the Gantt charts for Act_1, for two different starting resource availability vectors. Thick edges are used to indicate the time at which resources are claimed and released by the activity. The light gray area denotes that we have to wait until the resource becomes available, and the light yellow areas indicate that the resource is claimed but no action is being executed on it.

The timing semantics of an activity sequence is defined in terms of repeated matrix multiplication. Note that alternatively, the timing can also be computed by first composing all activities using the sequencing operator (Def. 3). The matrix multiplication is however more efficient, since each activity matrix has to be computed only once.

Lemma 9 ($(\max, +)$ dynamics of an activity sequence)**.** *Consider activities Act_1 and Act_2. Then $\boldsymbol{M}_{Act_1 \cdot Act_2} = \boldsymbol{M}_{Act_2} \otimes \boldsymbol{M}_{Act_1}$.*

Fig. 6 shows the Gantt chart induced by activity sequence $Act_1 \cdot Act_2 \cdot Act_3$. Note that activities are pipelined on the resources. For instance, Act_2 starts before Act_1 is fully completed.

C. Dispatching Activities

We use a non-deterministic finite state machine (FSM) to model all allowed (possibly infinite) activity sequences. These

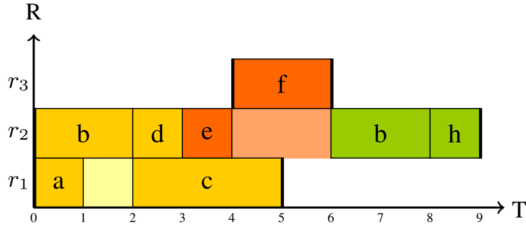


Fig. 6. Gantt chart of $Act_1 \cdot Act_2 \cdot Act_3$ given starting vector $\gamma_R = [0, 0, 0]^\intercal$.

activity sequences ensure that functional requirements are met, for instance related to enforcing product life cycles and ensuring safety aspects.

Definition 10 (Activity-FSM)**.** *An Activity-FSM F on $\mathcal{A}ct$ is a tuple $\langle L, \mathcal{A}ct, \delta, l_0 \rangle$ where L is a finite set of locations, $\mathcal{A}ct$ is a nonempty set of activities, $\delta \subseteq L \times \mathcal{A}ct \times L$ is the transition relation, and $l_0 \in L$ is the initial location. Let $l \overset{Act}{\to} l'$ be a shorthand for $(l, Act, l') \in \delta$.*

The timing of activities can be added to the Activity-FSM by adding the $(\max, +)$ matrix of each activity to the corresponding edges. From this automaton a $(\max, +)$ state space can be generated for performance analysis of the controller.

Definition 11 (Normalized $(\max, +)$ state space (adapted from [7]))**.** *Given Activity-FSM $\langle L, \mathcal{A}ct, \delta, l_0 \rangle$, resource set $\mathcal{R}$, and $(\max, +)$ matrix set $\{\boldsymbol{M}_{Act} \mid Act \in \mathcal{A}ct\}$, we define the normalized $(\max, +)$ state space $\langle C, c_o, \Delta \rangle$ as follows.*

- *Initial configuration $c_0 = \langle l_0, \mathbf{0} \rangle$.*
- *Set $C = L \times \mathbb{R}^{-\infty^{|\mathcal{R}|}}$ of configurations consisting of a location and a normalized resource availability vector.*
- *A labeled transition relation $\Delta \subseteq C \times \mathbb{R} \times \mathcal{A}ct \times C$ consisting of the transitions in the set $\{\langle\langle l, \gamma_R \rangle, \left\| \gamma'_R \right\|, Act, \langle l', \gamma'^{norm}_R \rangle\rangle \mid$ $(l, Act, l') \in \delta \wedge \gamma'_R = \boldsymbol{M}_{Act} \otimes \gamma_R\}$.*

Each state $\langle l, \gamma_R \rangle$ refers to both an FSM location l, and a resource availability vector γ_R. Consider an edge $\langle\langle l, \gamma_R \rangle, \left\| \gamma'_R \right\|, Act, \langle l', \gamma'^{norm}_R \rangle\rangle$. We start from state $\langle l, \gamma_R \rangle$, and execute the scenario on the edge $\langle l, l' \rangle$ in the FSM. $\left\| \gamma'_R \right\|$ denotes the transit time. The new state is $\langle l', \gamma'^{norm}_R \rangle$, where the new resource time stamp vector is computed as $\gamma'_R = \boldsymbol{M}_{Act} \otimes \gamma_R$, which is subsequently normalized. The state space records only the normalized resource availability vectors, since only the relative timing differences affect the future behavior, not their absolute offset.

Each reachable cycle in this state space allows for a periodic execution of the system. Each edge on this cycle is associated with a *transit time* corresponding to the activity duration. Let the transit time of a cycle be the sum of the transit time values of its edges. Then the *cycle mean* is equal to its transit time divided by the number of edges in the cycle. Both the best and worst case performance of the system can be found by looking at these cycles, using an maximum or minimum cycle mean algorithm [4], [7].

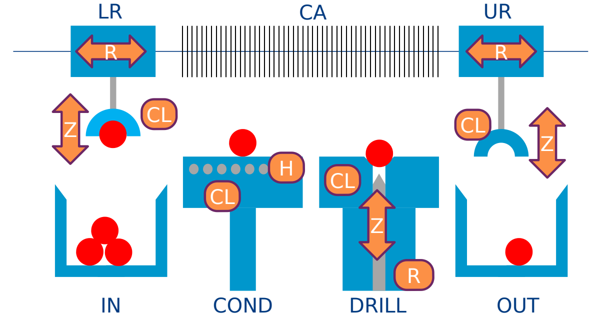


Fig. 7. Manufacturing system example (Twilight system) with two robots and two production stages.

III. Example: Twilight System

In this section we show how the modeling approach can be used to model and analyze a manufacturing system. As an example we take the Twilight system shown in Fig. 7, where balls are processed each following a given recipe. This manufacturing system is a simplification of the product handling model that has been created at ASML, using similar kinds of peripherals and resources.

A. Example Manufacturing System

Our example system contains four resources. First, there are two robots to transport balls; the load robot (LR) and the unload robot (UR). Each robot has a homing position; LR on the left corner, and UR on the right corner. The other two resources are processing stations, the conditioner (COND) to ensure a right ball temperature, and the drill (DRILL) to drill a hole in a ball. Both robots have three peripherals; a clamp (CL) to pick up and hold a ball, an R-motor (R) to move along the rail, and a Z-motor (Z) to move the clamp up and down. Since each robot can reach both processing stations, there is a collision area (CA). Both processing stations have a clamp peripheral. The conditioner has a heater (H), to heat a ball. The drill has an R-motor (R) to rotate the drill bit, and a Z-motor (Z) to move the drill bit up and down.

Each ball processed by the system follows the same life cycle. First, a ball is picked up at the input buffer by the load robot. Then it is brought to the conditioner and processed. Next, the item is transported by either one of the robots to the drill, where it is drilled. Finally, the drilled ball is transported to the output buffer.

B. Activities

In our system, there are two activities that process balls: Condition and Drill. For transportation of the balls, there are two types of activities: picking up a ball by a robot, and releasing a ball by a robot on a product location. The complete set of activities is shown in Table I.

Each activity is modeled formally by specifying the actions involved and the dependencies between these actions. As an example, consider activity LR_PickFromCond shown in Fig. 8, in which the load robot picks a ball from the conditioner.

In activity LR_PickFromCond, we use the special resource CA to model the physical collision area above COND and

TABLE I
SET OF ACTIVITIES FOR OUR EXAMPLE SYSTEM.

LR_PickFromInput	LR_PutOnDrill	UR_PutOnCond
LR_PutOnCond	UR_PickFromDrill	UR_PutOnOutput
LR_PickFromCond	UR_PickFromCond	Condition
LR_PickFromDrill	UR_PutOnDrill	Drill

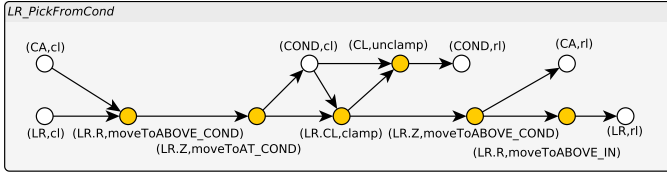


Fig. 8. Activity LR_PickFromCond.

DRILL. As long as one robot has claimed this resource, the other robot cannot enter. Robots always return to their safe home position before ending a robot activity. Using a homing position guarantees safety, but might not result in a throughput-optimal system. More refined activities allow more scheduling freedom by the controller, which can be used to improve the maximal achievable throughput. We will not consider such refinements here.

C. Allowed Activity Sequences

Given the system activities, we model which activity sequences are allowed. This is done using a set of requirements, modeled as automata, where the transitions are labeled with activity names. Multiparty synchronization is used, where execution of shared events is synchronous. This synchronization mechanism ensures that after composition, each requirement is still taken into account.

As mentioned before, each activity involves the transport or processing of a ball. In the model, we explicitly model the ball instances in the system by adding an identifier i in the the activity name suffix. An infinite product stream is simulated by using five ball instances (see Fig. 9), induced by the resource capacity of the system (see also [1]). Products enter the system in the order induced by their indices. Given an activity Act involving a product and set I of product identifiers, we define set $Act_* = \{Act_i \mid i \in I\}$.

For each ball we model the location in the system and the enabled activities, shown in Fig. 10. For instance, if a ball is at the drill (atDrill), the system can perform the Drill activity on this ball, or pick it up by one of the robots.

Each ball is required to follow the same life cycle. This requirement is modeled using the automaton shown in Fig. 11.

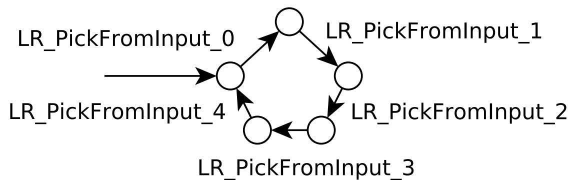


Fig. 9. Product order automaton.

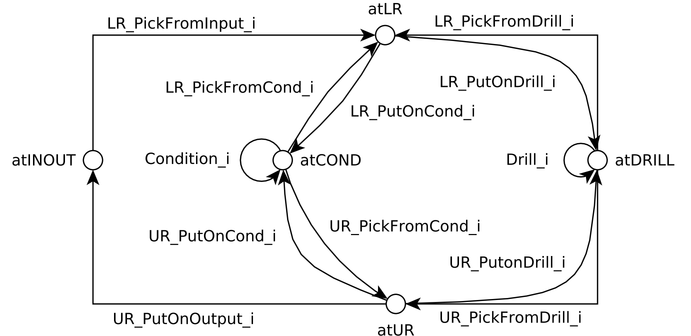


Fig. 10. Product location automaton.

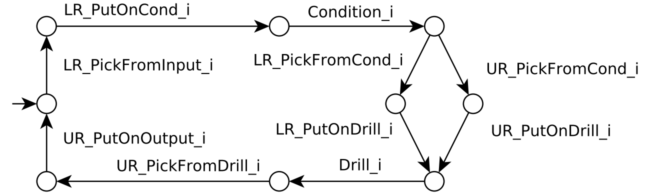


Fig. 11. Life cycle automaton.

Moves are explicitly encoded to ensure that balls always move forward in the system. In this way, we can find a meaningful minimal throughput guarantee in the analysis step. Note that there is still scheduling freedom which robot is used to transport a ball from the conditioner to the drill. This choice might have an impact on the overall system performance.

To avoid ball collisions, we add location state automata, shown in Fig. 12. These automata ensure that after picking up a ball by a robot, it must first be released before the next ball can be picked. In the same way, we avoid putting two balls on the conditioner or the drill.

Given the set of activities, and the requirements, we use supervisory controller synthesis [8], [9] to obtain an Activity-FSM of all allowed activity sequences. By using synthesis, the Activity-FSM is guaranteed to be deadlock-free and functionally correct with respect to the modeled requirements. The resulting Activity-FSM after synthesis is shown in Fig. 13.

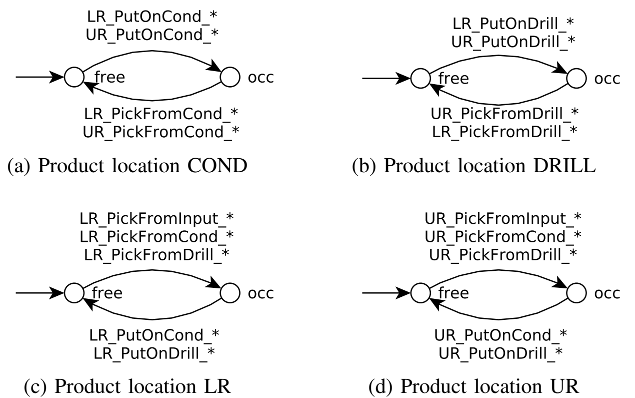


Fig. 12. Location state automata.

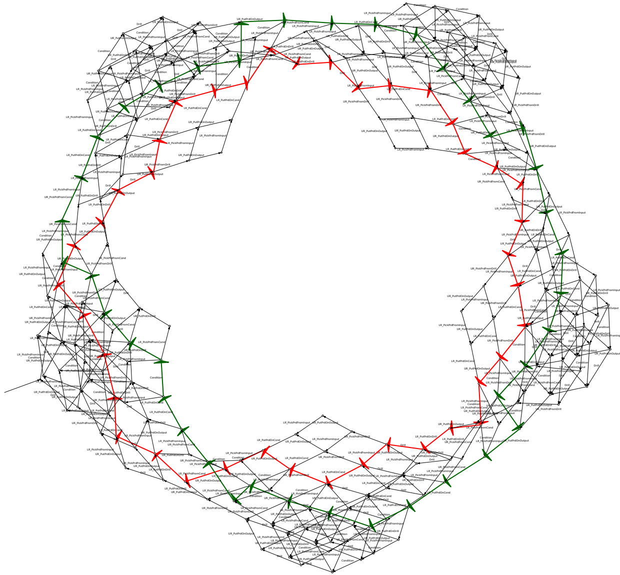


Fig. 13. Synthesized Activity-FSM for our example (245 locations and 510 transitions). The green cycle is an optimal dispatch sequence, the red cycle is a worst-case dispatch sequence.

Since requirements are modeled in a compositional way, it is also possible to use modular synthesis techniques [10], [11] for larger cases. The result is a set of supervisors that work in conjunction to enforce the complete set of requirements.

D. Analysis

To do performance analysis, we compute the matrices of the activities and then the normalized $(\max, +)$ state space. This state space contains 1633 states and 2894 edges. To find the best-case throughput of the system, we use the cycle analysis algorithm as described in Section II-C. This algorithm yields the optimal dispatch sequence for steady-state behavior, shown in green in Fig. 13.

E. Explicit Modeling using Finite Automata

We have also modeled the example system explicitly using finite state automata with timing, to verify the benefit of our modeling approach with respect to scalability. This benefit arises from the abstraction of the actions in activities. In the full model, there is explicit interleaving of all actions contained in activities. Note that the full model contains the same schedules, since we only unfold the interleaving of low-level actions.

Each activity is modeled as an automaton, and instantiated for each occurrence in the Activity-FSM with a unique identifier. Per resource we have an automaton to capture the precedence constraints among activities. For each resource, there is an automaton that ensures correct claiming and releasing. In this model, also multi-party synchronization is used. As a modeling tool we have used CIF3 [12].

Exploration of the full state space ran out of memory after an hour on an Intel E5-2630 CPU and 100GB of available virtual memory. We found out that the full state space contains at least 5 million states. This shows the huge state space reduction that can be achieved by the $(\max, +)$ state space, compared to a state space with full interleaving of actions. The experiment shows the benefit of our formal modeling approach, where analysis can be done within a few seconds.

IV. Industrial Application

At ASML, the modeling formalism is used as a semantic underpinning of a DSL to express part of the system behavior. The DSL allows domain engineers to describe the system in terms of resources, peripherals, actions, and activities. An important subset of the available actions are determined by symbolic positions and motion paths of robot resources. For example, two robot arms have been modeled that have 3 axes, around 50 symbolic positions each, and around 300 motion paths between symbolic positions resulting in a large set of possible peripheral actions. Using this system model, various activity sequences can be analyzed in terms of performance. Also, the impact of individual action timings on machine throughput is analyzed. This timing analysis is enabled by the $(\max, +)$ semantics and the available analysis techniques in this domain that can easily be applied to the system models. The current focus is on obtaining a complete specification for nominal behavior in the product handling part of the machine.

There is a formal specification model of the product logistics [1], that can be linked to the system model. However, synthesis of the Activity-FSM is not feasible due to scalability issues. Therefore, an important current research topic is modular synthesis of the Activity-FSM to improve scalability.

V. Related Work

Scenario-Aware Data Flow (SADF) [7] has a similar separation of concerns with respect to functionality and timing. This formalism uses the same model of computation, $(\max, +)$ algebra, to perform throughput analysis. Compared to SADF, our formal modeling approach allows modular specification of both the activities and the requirements to be imposed on the ordering. Another advantage of our approach is the ability to synthesize the FSM containing all allowed activity sequences. These two advantages also apply with respect to $(\max, +)$ automata. It is possible to convert a model in our formalism to an FSM-SADF model. As studied in [13], each activity can be mapped onto an SDF scenario, and the Activity-FSM corresponds to the FSM in FSM-SADF. Note that an SDF scenario is more general, since it can also contain cycles.

Other well known formalisms used for modeling and performance analysis of manufacturing systems are timed automata [14], timed Petri nets [15], and job shop scheduling with precedence constraints [16]. Timed automata extended with game theory, timed games [17], allow synthesis of a controller ensuring safe behavior and reaching a final state eventually. Timed Petri nets allow performance analysis by associating delay bounds with each place in the net [18]. In both formal models, specification of timing and functionality aspects is not compositional. Instead, a system with actions, activities and resources is typically specified as described in Section III-E, which supports the claim of a similar type of

scalability issues. In job shop scheduling, finding an optimal controller is typically considered as a constraint satisfaction problem. Here, all dependencies on job (activity) level and operation (action) level are encoded using constraints. In this formulation, there is also no separation of concerns with respect to functionality and timing.

Modeling system operations as graphs is also done in other domains. For instance in real-time systems [19], where also deadlines and bounded inter-arrival times of actions play a role. However, activities are assumed to be independent, which means that there is no way to specify dependencies among the activities, which we do by means of the Activity-FSM.

Our framework also resembles partial-order automata [20], where each transition corresponds to a set of partial-ordered traces. In our case, each activity transition in the Activity-FSM also corresponds to a DAG describing multiple allowed traces. However, in partial-order automata, after each transition follows a synchronization point, which means that there is no weak-sequencing.

VI. Conclusion

This paper introduces a new formal modeling approach that allows compositional specification of both functionality and timing of manufacturing systems. In the approach, behavior in the system is abstracted using the concept of activities, and the controller choices reside on this abstracted level. This abstraction leads to a concise specification, and results in a smaller state space for synthesis and analysis, compared to a state space with full interleaving of low-level actions. Resource management is handled at the lower level, without explicit interaction with the controller. The semantics of the model are expressed in the $(\max, +)$ domain, enabling the use of existing performance analysis techniques to find an optimal activity ordering. Our approach is illustrated on an example manufacturing system and an industrial case study is given.

There are a number of next steps to extend the approach. First, we want to look into the extension of dealing with exceptional behavior and external disturbances. In such a setting, the Activity-FSM will need to be extended with uncontrollable transitions that capture this uncontrollable behavior. Second, specification of functional requirements is now modeled directly using automata. In principle however, there are many other formalisms that may be better suited to specify requirements. For example, extended finite automata [21] can be used, where data variables are added to finite automata. This enables the use of state-based requirements [22], which allows requirements to refer to states of other automata. Third, in the industrial use of the modeling formalism, there are scalability challenges in synthesis of the Activity-FSM. Therefore, we are investigating several approaches to reduce the state space that needs to be explored using partial order reduction techniques, and the use of modular synthesis techniques.

Acknowledgment

This research is supported by the Dutch Technology Foundation STW, carried out as part of the Robust Cyber-Physical Systems (RCPS) program, project number 12694.

References

[1] B. van der Sanden *et al.*, “Modular model-based supervisory controller design for wafer logistics in lithography machines,” in *18th ACM/IEEE Int. Conf. on Model Driven Engineering Languages and Systems, MoDELS 2015, Ottawa, ON, Canada*, 2015, pp. 416–425.

[2] F. Baccelli *et al.*, *Synchronization and linearity: an algebra for discrete event systems*. John Wiley and Sons, 1992.

[3] S. Gaubert, “Performance evaluation of (max,+) automata,” *IEEE Trans. Autom. Control*, vol. 40, no. 12, pp. 2014–2025, Dec 1995.

[4] A. Dasdan, “Experimental analysis of the fastest optimum cycle ratio and mean algorithms,” *ACM Trans. Design Automation of Electronic Systems*, vol. 9, no. 4, pp. 385–418, 2004.

[5] A. Rensink and H. Wehrheim, *Weak Sequential Composition in Process Algebras*. Springer Berlin Heidelberg, 1994, pp. 226–241.

[6] M. Geilen, “Synchronous dataflow scenarios,” *ACM Trans. Embed. Comput. Syst.*, vol. 10, no. 2, pp. 16:1–16:31, Jan. 2011.

[7] M. Geilen and S. Stuijk, “Worst-case performance analysis of Synchronous Dataflow scenarios,” *2010 IEEE/ACM/IFIP Int. Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, no. C, pp. 125–134, 2010.

[8] P. J. G. Ramadge and W. M. Wonham, “Supervisory control of a class of discrete event processes,” *SIAM J. Control Optim.*, vol. 25, no. 1, pp. 206–230, 1987.

[9] ——, “The control of discrete event systems,” *Proc. of the IEEE*, vol. 77, no. 1, pp. 81–98, 1989.

[10] K. Schmidt and C. Breindl, “Maximally permissive hierarchical control of decentralized discrete event systems,” *IEEE Trans. Autom. Control*, vol. 56, no. 4, pp. 723–737, April 2011.

[11] L. Feng and W. Wonham, “Supervisory control architecture for discrete-event systems,” *IEEE Trans. Autom. Control*, vol. 53, no. 6, pp. 1449–1461, July 2008.

[12] D. A. van Beek *et al.*, “CIF 3: Model-based engineering of supervisory controllers,” in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Comput. Sci, E. Ábrahám and K. Havelund, Eds. Springer Berlin Heidelberg, 2014, vol. 8413, pp. 575–580.

[13] J. Bastos *et al.*, “Modeling resource sharing using FSM-SADF,” in *13. ACM/IEEE Int. Conf. on Formal Methods and Models for Codesign, MEMOCODE 2015, Austin, TX, USA, September 21-23, 2015*, 2015, pp. 96–101.

[14] R. Alur and D. L. Dill, “A theory of timed automata,” *Theoretical Computer Science*, vol. 126, no. 2, pp. 183 – 235, 1994.

[15] B. Berthomieu and M. Diaz, “Modeling and verification of time dependent systems using time petri nets,” *IEEE Trans. Softw. Eng.*, vol. 17, no. 3, pp. 259–273, Mar 1991.

[16] A. Garrido *et al.*, “Heuristic methods for solving job-shop scheduling problems,” in *Proc. ECAI-2000 Workshop on New Results in Planning, Scheduling and Design*, 2000, pp. 44–49.

[17] G. Behrmann *et al.*, *Computer Aided Verification: 19th Int. Conf., CAV 2007, Berlin, Germany, July 3-7, 2007. Proc.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, ch. UPPAAL-Tiga: Time for Playing Games!, pp. 121–125.

[18] H. Hulgaard and S. M. Burns, *Computer Aided Verification: 7th Int. Conf., CAV '95 Liège, Belgium, July 3–5, 1995 Proc.* Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, ch. Efficient timing analysis of a class of Petri nets, pp. 423–436.

[19] M. Stigge *et al.*, “The digraph real-time task model,” in *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, April 2011, pp. 71–80.

[20] G. v. Bochmann *et al.*, *Testing Systems Specified as Partial Order Input/Output Automata*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 169–183.

[21] M. Skoldstam, K. Åkesson, and M. Fabian, “Modeling of discrete event systems using finite automata with variables,” in *Decision and Control, 2007 46th IEEE Conf. on*, Dec 2007, pp. 3387–3392.

[22] J. Markovski *et al.*, “A state-based framework for supervisory control synthesis and verification,” in *Decision and Control (CDC), 2010 49th IEEE Conf. on*, Dec 2010, pp. 3481–3486.