Conversion of LSAT behavioral specifications to automata

Sander Thuijsman and Michel Reniers

Abstract

The Logistics Specification and Analysis Tool (LSAT) is a model-based engineering tool used for manufacturing system design and analysis. Using a domain specific language, a system can be specified in LSAT. In this paper, a conversion method is presented to obtain the system behavior of an LSAT specification in automata structure.

I. INTRODUCTION

The Logistics Specification and Analysis Tool (LSAT)[1] is a workbench for model-based system design and analysis to enable concept flexible design of manufacturing systems. LSAT enables rapid design-space exploration of supervisory controllers that orchestrate the behavior of these systems. The foundations of LSAT have been described in literature, without explicitly mentioning the tool itself. [1] provides an overview in which a number of LSAT's functions are used; they consider the xCPS flexible manufacturing system [2]. [1] shows how high-level activities can be used that consist out of low-level actions in order to describe the system behavior in a compact manner. [1], [3], [4] show how a Finite State Automaton (FSA) can be constructed on activity-level, which is much smaller than the would-be state space on action-level, to express the system behavior. Requirements are expressed on activity-level, and supervisor synthesis is applied in order to find the *maximally permissive*, *safe*, and *nonblocking* behavior [5]. Timing characteristics of an activity can be captured using (max,+) algebra [6]. Using these characteristics, a (max,+) statespace can be constructed. A throughput or makespan optimal controller of the system can be found in the (max,+) statespace using optimal cycle ratio algorithms [7], [8]. Partial-order reduction is applied to improve the computational performance of the (max,+) analysis in [9]. To improve system performance, a critical path technique [10] can be applied to identify the system's bottlenecks. During the engineering process, LSAT has the ability to supply several graphical visualizations, such as Gantt charts, movement plots, probability distributions, and system architecture diagrams.

Except for supervisor synthesis, all mentioned model-based system engineering procedures can be performed using LSAT. Synthesis is performed using CIF [11][2], which is added to LSAT through a plugin. CIF is an automata-based language and tool set used for modeling and synthesizing supervisory controllers.

In previous work supervisor synthesis and timing analysis was performed on activity-level [3], [4]. The abstraction from action-level to activity-level can be beneficial, as performing these computations on action-level can result in scalability issues [12]. However, some computations still require an explicit behavioral model at action-level. For example, behavioral requirements may be given on action-level. It is then impossible to (directly) perform supervisor synthesis or formal verification using these requirements without considering the action-level behavior to at least some extent. In this work we elaborate on the semantics of the specification of a manufacturing system in LSAT, and transform the action-level

[1]LSAT is developed by ASML, TNO-ESI and Eindhoven University of Technology. The tool and documentation can be found at https://esi.nl/research/output/tools/lsat.

[2]CIF is developed by Eindhoven University of Technology. The tool and documentation can be found at https://cif.se.wtb.tue.nl.

behavior of such a specification to automata. Such a model is used to understand the action-level behavior of an LSAT specification and is a prerequisite for developing automata-based supervisor synthesis or verification techniques for these systems.

The paper is structured as follows. In Section II we explain specification in LSAT, followed by an elaboration on the interacting system components. Section III starts with preliminaries of automata, and then uses the specification elements of Section II to construct sets of automata that describe the behavior of an LSAT specification. Conclusions are given in Section IV.

II. LSAT SPECIFICATION

In this section, we first introduce the specification elements that comprise an LSAT specification. Then, an interpretation of the control system architecture is provided, that uses the conceptual elements of the specification.

A. Specification elements

An LSAT specification contains the following elements, these definitions are strongly based on [4]:

1) Peripherals: A set of peripherals $\mathcal{P}$ is defined. The set of peripherals is split into two disjoint sets of either *unmovable* $p_u \in \mathcal{P}_u$, or *movable* $p_m \in \mathcal{P}_m$ peripherals. A movable peripheral p_m contains a set of positions L_p. For an unmovable peripheral no positions are defined.

2) Resources: Each peripheral p is contained within exactly one resource $r \in \mathcal{R}$. We assume a function $R : \mathcal{P} \to \mathcal{R}$, s.t. $R(p)$ is the resource containing p.

3) Actions: An unmovable peripheral contains a set of actions $\mathbb{A}_u \subseteq \mathbb{A}$ it can execute. For every action of an unmovable peripheral the time it takes to execute the action is specified. This amount of time may be deterministic or stochastic in the form of a normal, triangular, or PERT distribution [13].

For a movable peripheral movement actions $\mathbb{A}_m \subseteq \mathbb{A}$ are defined that are directed movements between its locations. A movable peripheral can move between all or a subset of its positions. For the movement actions we assume functions $L_s : \mathbb{A}_m \to L_p$ and $L_t : \mathbb{A}_m \to L_p$ that respectively define the source and target location of the movement of the action. A movement has a speed profile, which is either a second- or third-order profile, and an optional settling time. Using the speed profile, the time of the movement can be determined [14].

An action can be performed on exactly one peripheral. We assume a function $A : \mathcal{P} \to 2^{\mathbb{A}}$ indicating the actions that are performed by a peripheral.

4) Activities: An activity $Act \in \mathcal{A}$ is a DAG $(N, \to)$, consisting of a set nodes N and a set of dependencies $\to \subseteq N \times N$. We assume a mapping function $M : N \to (\mathbb{A} \times \mathcal{P}) \cup (\mathcal{R} \times \{rl, cl\})$, which associates a node to either a pair (a, p) referring to execution of action a on peripheral p; 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. All nodes mapped to the same peripheral are sequentially ordered to avoid self concurrency. All action nodes are preceded by a claim node and succeeded by a release node of the corresponding resource. Resources are claimed and released not more than once within an activity. For the corresponding resource, a claim node is always succeeded by a release node, and a release node is always preceded by a claim node. We assume a function $R : \mathcal{A} \to 2^{\mathcal{R}}$, s.t. $R(Act)$ is the set of resources that Act uses.

5) Dispatching sequences: We define a finite activity sequence ω as an ordered list of activities, $\omega = Act_1; Act_2; ...; Act_n$. Sequences can be concatenated using the ';' operator; $\omega = \omega_1; \omega_2$. ω_1 and ω_2 are then called subsequences. ε is the empty sequence, so $\omega; \varepsilon = \omega$. $\omega(i)$ denotes the *i'th* activity in the sequence (in the case of the given sequence that is Act_i). We use notation $\omega(i : j)$ to denote the subsequence that goes from the *i'th* to the *j'th* activity in the sequence. $|\omega|$ denotes the length of a sequence. The set of prefix sequences is given by $\text{prefix}(\omega) = \{\varepsilon\} \cup \{\omega(1 : i)|1 \le i \le |\omega|\}$. We use notation ω^n to denote n concatenated subsequences ω: $\omega^{n+1} = \omega^n; \omega$, and $\omega^0 = \varepsilon$.

A *dispatching sequence seq* is given as a finite activity sequence, appended by a finite activity sequence that infinitely repeats; $seq = \omega_1; (\omega_2)^\infty$. Such a structure with a transient phase followed by a periodic phase is consistent with the operational semantics of synchronous data flow graphs [15]. LSAT uses these semantics for the timing optimization synthesis algorithms. Note that ω_1 and ω_2 may or may not be empty, and $\varepsilon^\infty = \varepsilon$. The prefix operation for a dispatching sequence follows: prefix$(\omega_1; (\omega_2)^\infty)$ =prefix$(\omega_1) \cup \{\omega_1; (\omega_2)^n; \omega_p | n \in \mathbb{N}_{\geq 0}, \omega_p \in$prefix$(\omega_2)\}$

A single dispatching sequence can directly be specified, or a set of possible dispatching sequences can be provided by a (network of) FSA. These FSA have the activities' names as event labels and their synchronous language defines possible dispatching sequences, this is further discussed in Section III-G.

B. System components

The behavior of a manufacturing system specified in LSAT can be modeled by the interacting components that we introduce in this section. We introduce some vocabulary of these components and their behavior, which we will describe formally in Section III. Figure 1 shows an illustration of the system's components.
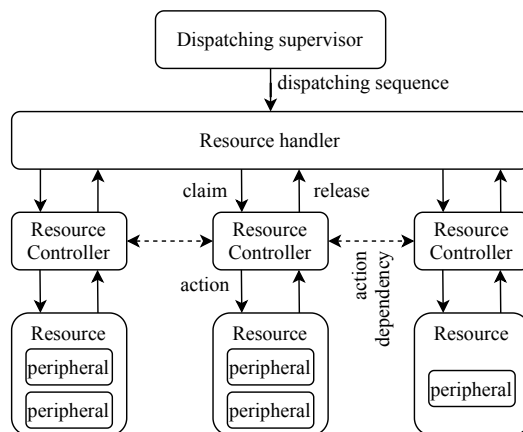

Fig. 1.   Components that model the behavior of an LSAT specification

The *dispatching supervisor* provides a dispatching sequence to the *resource handler*. The resource handler acts as a FIFO queue and will claim each resource for the activity that is to use it next in the dispatching sequence, when all incoming dependencies of the particular claim node in the activity are met. While the resource is claimed, the *resource controller* executes the actions on the peripherals of the resource that are associated with the action nodes in the activity model. The resource is released once all incoming dependencies to the release node are met, making the resource available for claiming by a next activity. There is direct communication between the resource controllers so that dependencies between actions on different resources can (and will) be enforced.

III. BEHAVIORAL MODEL

In this section we use automata to give a formal behavioral description of the system components introduced in Section II. In Section III-A we will provide some preliminaries on automata. In Section III-B we will discuss instantiation of Activities and the instantiated dispatching sequence. In the subsections thereafter we will provide definitions for automata describing the different elements of the system. In Section III-G the combined behavior of the interacting automata is described. First, we consider an untimed description of the system behavior; actions occur in an instantaneous and discrete manner in the order as defined by the dependencies between the action nodes in the activities. An extension of the behavioral description that includes continuous characteristics is discussed in Section III-H.

Automata were chosen for this behavioral definition, rather than for example Petri Nets, because there is already existing integrated combined functionality between LSAT and CIF [11]. CIF is an automata-based language and toolset.

A. Preliminaries on Automata

The behavioral description might require automata that consist out an infinite amount of states, as shown in Section III-B. Therefore we use *infinite state automata* in our description, rather than more commonly used finite state automata. This means that synthesis or verification can not directly be applied to these models. However, the behavioral semantics are still clear. The models can act as a baseline to show behavioral equivalence for synthesis or verification techniques that are applied to finite behavioral descriptions.

An infinite state automaton is a *4-tuple* $A = (X, \Sigma, T, X_0)$, where X is the set of states, Σ is the set of events, $T \subseteq X \times \Sigma \times X$ is the set of transitions, and $X_0 \subseteq X$ is the set of initial states. Each of these sets may be finite or infinite [16]. Σ^* is the Kleene-closure of Σ [5]. We use notation $A.X$ to refer to states X in automaton A.

We use notation $x_s \xrightarrow{\sigma} x_t$ when there exists a transition $(x_s, \sigma, x_t) \in T$. We extend this notation to strings $w \in \Sigma^*$; $x_s \xrightarrow{w\sigma} x_t$ if $x_s \xrightarrow{w} x_i$ and $x_i \xrightarrow{\sigma} x_t$ for some intermediate state $x_i \in X$, and $x \xrightarrow{\varepsilon} x$ for all $x \in X$. The language of an automaton is given by the set of all possible strings that can be executed from some initial state: $L(A) = \{w | x_0 \xrightarrow{w} x_t, x_0 \in X_0, x_t \in X\}$. ε is the empty string.

The *synchronous composition* [5] of automata A_1 and A_2 is denoted as $A_1 || A_2$, which creates a new automaton that contains the combined behavior of A_1 and A_2. In this composition, the automata can independently execute their unique events, and shared events are synchronized in lock-step synchronization. Synchronous composition is associative and commutative, and therefore extends to a set of automata; $||(\{A_1, A_2, ..., A_n\})$ denotes $(A_1 || A_2 || ... || A_n)$.

B. Activity instantiation

Two instances of the same activity can be active at the same time. Let's consider the activity Act given in Figure 2, and a dispatching sequence $seq = Act; (Act)^\infty$. Let us refer to the first instance of Act in this sequence as Activity instance Act^1, and the second instance as Act^2. It is possible that Act^1 performs the a action on peripheral $p1$ of $R1$ and releases $R1$, so that Act^2 can claim this resource and perform the a action, before Act^1 has performed the b action. So, multiple instances of the same activity can be executed concurrently.
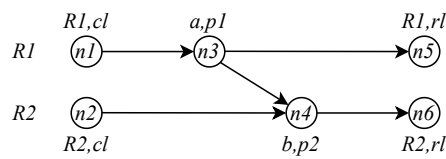
Fig. 2. Example Activity *Act* DAG

The number of concurrent activities of the same instance is unbounded. Continuing the previous example, it is possible for the activity instance Act^n to perform the a action before any activity instance Act^m, with $m < n$, has performed action b. Essentially, the amount of a's that have been performed, has an influence on the amount of b's that still may be performed. n may be of an arbitrarily large size. This means that (for some combinations of dispatching sequences and activities) it is required to keep track of an arbitrarily large (unbounded) amount of activity instances. For each activity instance an automaton is used to track its progress, so there might be infinitely many of such automata. Also in dispatching these (each time new) activity instances will occur, leading to automata with an infinite amount of transitions and states. In conclusion, we allow the behavioral description to use an infinite amount of states and/or an infinite amount of automata.

C. Used components

To construct the system behavior, we will make models of the above descriptions that are relevant to a given dispatching sequence. E.g., peripherals that are not used, or activity instances that do not appear in the instantiated dispatching sequence are also not included in the model. For $seq = \omega_1; (\omega_2)^\infty$, we define the set of used activities $\mathcal{A}_\in = \{Act \in \mathcal{A} | \exists i : \omega_1(i) = Act \vee \exists j : \omega_2(j) = Act\}$ used resources $\mathcal{R}_\in = \{r \in \mathcal{R} | r \in R(Act), Act \in \mathcal{A}_\in\}$, and used peripherals $\mathcal{P}_\in = \{p \in \mathcal{P} | M(n) = (a, p), n \in N, (N, \rightarrow) \in \mathcal{A}_\in\}$.

For any activity that appears in the ω_2 part of the dispatching sequence we model an infinte amount of activity instances. For any activity that appears in the ω_1, but not the ω_2, part of the dispatching sequence we model the amount of activity instances as the number of times it appears in the dispatching sequence. The set of used activity instances is given as $\tilde{\mathcal{A}}_\in = \{Act^j | Act \in \mathcal{A}, j \in \mathbb{N}_{>0}, 1 \le j \le |\{i \in \mathbb{N}_{>0} | \omega_1(i) = Act\}|\} \cup \{Act^j | Act \in \mathcal{A}, \exists(i \in \mathbb{N}_{>0}) : \omega_2(i) = Act, j \in \mathbb{N}_{>0}\}$.

To denote an instance of Act, without the explicit instance number, we write $\tilde{Act}$.

D. Resource handler

The resource handler allows a resource to be claimed by one activity instance at a time, and claims the resources for the activity instances as the respective activities appear in the dispatching sequence. We capture these two functionalities in two seperate sets of automata: *availability automata* and *claiming automata*.

1) Availability automata: Each resource can only be claimed by one activity instance at a time; claim and release actions for this resource must occur in alternating manner. To model this behavior, we construct a *resource availability automaton A_r* for each resource $r \in \mathcal{R}_\in$ as follows: $A_r = (X, \Sigma, T, X_0)$, where:

- $X = \{released, claimed\}$,
- $\Sigma = \{(\tilde{Act}, cl, r), (\tilde{Act}, rl, r) | \tilde{Act} \in \tilde{\mathcal{A}}_\in, R(Act) \ni r\}$
- $T = \{(released, (\tilde{Act}, cl, r), claimed) | \tilde{Act} \in \tilde{\mathcal{A}}_\in, r \in R(Act)\} \cup \{(claimed, (\tilde{Act}, rl, r), released) | \tilde{Act} \in \tilde{\mathcal{A}}_\in, r \in R(Act)\}$,
- $X_0 = \{released\}$.

Initially each resource is released. It can be claimed by any activity instance, by event $(\tilde{Act}, cl, r)$. After which, it can only be claimed again after it has been released; $(\tilde{Act}, rl, r)$. The releasing will occur as defined by the Activity automata, discussed later in Section III-E.

Example: An availability automaton for resource $R1$ is given in Figure 3.
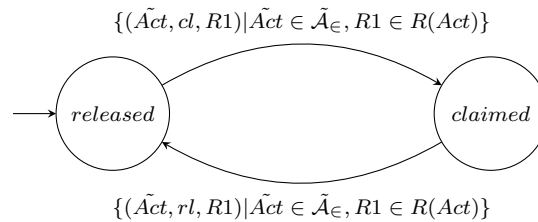

Fig. 3. Availability automaton A_{R1} for resource $R1$

2) Claiming automata: For each resource r we can define a *resource claiming automaton C_r*, that claims the particular resource in the order that it is used by the activity instances in the dispatching sequence. We introduce a function ρ_r that reduces a sequence so that it only contains the activity

instances that use the particular resource r, inductively defined as follows:

$$\rho_r(\varepsilon) = \varepsilon$$

$$\rho_r(Act; \omega) = \begin{cases} Act; \rho_r(\omega) & \text{if } r \in R(Act) \\ \rho_r(\omega) & \text{if } r \notin R(Act) \end{cases}$$

Given a dispatching sequence $seq = \omega_1; (\omega_2)^\infty$, we define the resource claiming automaton for resource r as follows: $C_r = (X, \Sigma, T, X_0)$, where:

- $X = $ prefix(seq_r), with $seq_r = \rho_r(\omega_1); (\rho_r(\omega_2))^\infty$
- $\Sigma = \{(\tilde{Act}, cl, r) | \tilde{Act} \in \tilde{\mathcal{A}}_\in, R(Act) \ni r\}$,
- $T = \{(\omega_x, (Act^j, cl, r), \omega_x; Act) | \omega_x \in X, \omega_x; Act \in X, j = 1 + |\{i \in \mathbb{N}_{>0} | \omega_x(i) = Act\}|\}$
- $X_0 = \{\varepsilon\}$.

The states X are defined by the sequence of activity instances that have been performed on resource r. The set of events Σ, are the claim actions of all activity instances that use resource r that appear in this sequence. Each time, the claim action of the first activity in this sequence transitions to the next state. Initially, all claim actions still need to be performed, so the empty sequence of activities that use resource r is the initial state. The instance number of the activity instance (j) is incremented by one each time the respective activity appears. For this, the amount of times the activity appears in the sequence of activities that have been performed thus far in the particular state is used.

Example: Given dispatching sequence $seq = Act_A; (Act_B; Act_C)^\infty$, where activity Act_A uses resource $R1$; Act_B uses $R1$ and $R2$; Act_C uses $R2$. Applying the definition above provides the two claiming automata shown in Figure 4.
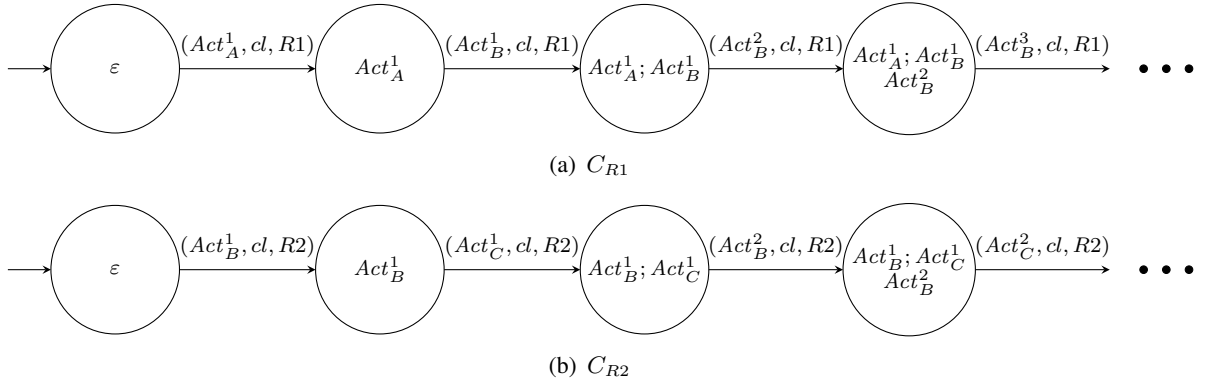

(a) C_{R1}

(b) C_{R2}

Fig. 4. Claiming automata, example

E. Resource controller

The resource controllers receive claim commands from the resource handler. They execute the actions as defined in the respective activity. The resource controllers communicate with each other when dependencies between them are defined in the activity definitions. The behavior of the resource controller can be found by converting the activities (given as DAG) into automata.

For a given DAG $Act = (N, \rightarrow)$, we define a *postset* $N' \subseteq N$ as a set of nodes in N that can be reached by iteratively removing nodes and their respecting outgoing dependencies that have no incoming dependencies in Act. Given $Act = (N, \rightarrow)$, N' is a *postset* of N if for all n in $N \setminus N'$: whenever $(m, n) \in \rightarrow$, then $m \notin N'$ (of all nodes that are removed from N to construct N', all their predecessor nodes are removed). Note that N and $\emptyset$ are postsets of N.

The Activity automaton $B_{\tilde{Act}}$ for activity instance $\tilde{Act}$ of activity $Act = (N, \rightarrow)$ can be given as follows: $B_{\tilde{Act}} = (X, \Sigma, T, X_0)$, where:

- $X = \{N'|N'$ is a postset of $N\}$,
- $\Sigma = \{(\tilde{Act}, a, p)|(a, p) \in N\}$,
- $T = \{(N', (\tilde{Act}, a, p), N' \setminus \{n\})|M(n) = (a, p), N' \in X, n \in N'\}$,
- $X_0 = \{N\}$.

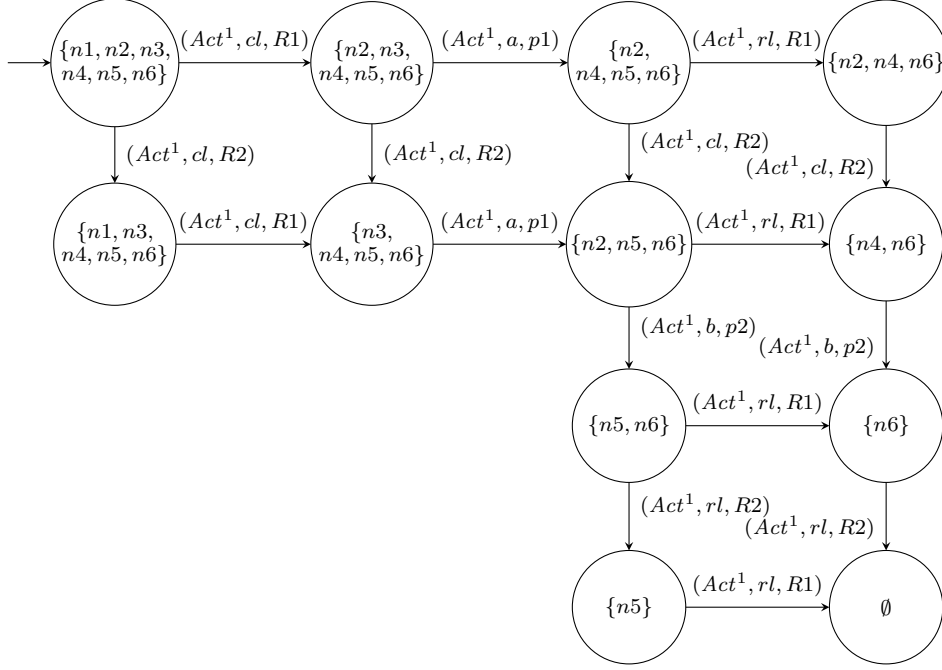*Example:* In Figure 5 the activity automaton for the DAG of Figure 2 is shown.

Fig. 5.   Example Activity instance automaton for Activity Act

F. Peripherals

The system contains a set op peripherals $\mathcal{P}$ that execute actions. To each peripheral p a state is associated, which is defined by the last action that is executed on the peripheral. E.g., for a lamp peripheral, the state of the lamp is turned on if the last executed action was turning on the lamp. The peripheral automata only contain actions that are used by one of the activities in the dispatching sequence.

A peripheral automaton for an unmovable peripheral p (with $p \in \mathcal{P}_u$) is given as follows. $Q_p = (X, \Sigma, T, X_0)$, where:

- $X = \{a|Act \in \mathcal{A}_\in, Act = (N, \rightarrow), n \in N, M(n) = (a, p)\}$
- $\Sigma = \{(\tilde{Act}, a, p)|\tilde{Act} \in \tilde{\mathcal{A}}_\in, Act = (N, \rightarrow), n \in N, M(n) = (a, p)\}$,
- $T = \{(x_s, (\tilde{Act}, a, p), a)|x_s \in X, (\tilde{Act}, a, p) \in \Sigma\}$,
- $X_0 = X$.

A peripheral automaton for a movable peripheral p (with $p \in \mathcal{P}_m$), is given as follows. $Q_p = (X, \Sigma, T, X_0)$, where:

- $X = \{\{L_s(a)\} \cup \{L_t(a)\}|\tilde{Act} \in \tilde{\mathcal{A}}_\in, Act = (N, \rightarrow), n \in N, M(n) = (a, p)\}$,
- $\Sigma = \{(\tilde{Act}, a, p)|\tilde{Act} \in \tilde{\mathcal{A}}_\in, Act = (N, \rightarrow), n \in N, M(n) = (a, p)\}$,
- $T = \{(L_s(a), (\tilde{Act}, a, p), L_t(a))|(\tilde{Act}, a, p) \in \Sigma\}$,
- $X_0 = X$.

Note that we use the same action events as for the activity automata, so that the peripheral automata and activity automata synchronize. This means multiple transitions can be added for the same action, as they appear in different activity automata (e.g., $\sigma_1 = (Act^1_A, a_1, p_1), \sigma_2 = (Act^2_A, a_1, p_1)$). Note that

in the movable peripheral definitions sometimes transitions are disabled, these will also not be present in the FSA. In an LSAT specification, no initial states or positions are specified for the peripherals. As such, any state of the peripheral automata may be an initial state.

Example: In Figure 6(a) an automaton corresponding to an unmovable peripheral p_u on which activity instance Act_A^1 can execute actions a and b is shown. In Figure 6(b) an automaton is given that corresponds to a movable peripheral p_m with positions *left, middle,* and *right*, that can move between *left* and *middle*, and between *middle* and *right*. These actions are used by activity instance Act_B^1.
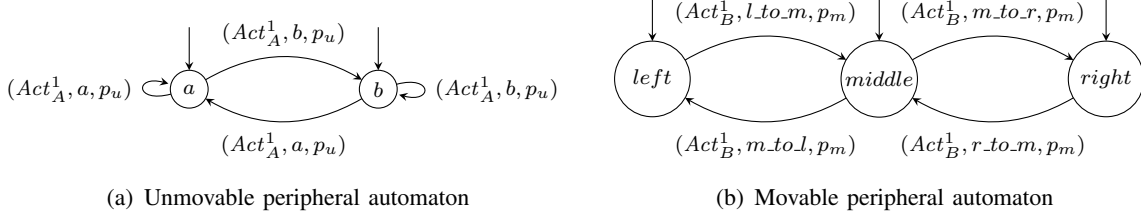

(a) Unmovable peripheral automaton

(b) Movable peripheral automaton

Fig. 6. Peripheral automata

G. Combined system behavior

For a given dispatching sequence seq, we can construct the automata described above, their synchronous composition of the relevant components (that appear in seq) describes the system behavior for the sequence in automaton M^{seq}:

$$M^{seq} = \overline{||}(\{A_r | r \in \mathcal{R}_\in\} \cup \{C_r | r \in \mathcal{R}_\in\} \cup \{B_{\tilde{Act}} | \tilde{Act} \in \tilde{\mathcal{A}}_\in\} \cup \{Q_p | p \in \mathcal{P}_\in\})$$

As per the definitions above, M^{seq} may be given in a finite or infinite manner, depending whether seq has an infinite part or not.

It is also possible to give a (network of) FSA that describes a set of possible dispatching sequences. The alphabet of these FSA consists of the activities' names. We refer to the the synchronous composition of these FSA as FSA D. The language of this automaton is given by a set of words $w \in L(D)$. Such words can be seen as dispatching sequences, with the activities in the same order in the sequence as the word. We then write $w \equiv seq$. As such, a set of dispatching sequences Seq is found for the language $L(D)$. A set of sequences Seq is considered *complete* for a dispatching FSA D iff: (1) there does not exist a word in the language of D that is not a prefix of any sequence in seq; $\forall w \in L(D)$: there exists $seq_p \in \text{prefix}(seq)$ with $seq \in Seq$ such that $w \equiv seq_p$, and (2), there does not exist a prefix of a sequence seq in Seq that is not in the language of D; $\forall seq \in Seq(\forall seq_p \in \text{prefix}(seq))$: there exists $w \in L(D)$ such that $w \equiv seq_p$.

For each $seq \in Seq$, which may or may not be infinitely many, the automaton M^{seq} can be constructed as described in the previous sections. The combined behavior of all possible dispatching sequences is then described in infinite state automaton $M^{Seq} = (X, \Sigma, T, X_0)$, where:

- $X = \{x^{seq} | x \in M^{seq}.X, seq \in Seq\}$
- $\Sigma = \{\sigma | \sigma \in M^{seq}.\Sigma, seq \in Seq\}$
- $T = \{(x_s^{seq}, \sigma, x_t^{seq}) | (x_s, \sigma, x_t) \in M^{seq}.T, seq \in Seq\}$
- $X_0 = \{x_0^{seq} | x_0 \in M^{seq}.X_0, seq \in Seq\}$

Essentially the automaton M^{Seq} defines the union of the behavior of all separate automata M^{seq}. So, $L(M^{Seq}) = \bigcup_{seq \in Seq} L(M^{seq})$.

The authors note that complete sets Seq can be constructed in multiple ways. For any pair of sets (Seq_1, Seq_2), where each set of sequences is complete for automaton D, the resulting automaton M^{Seq} has the same language. Because for any $seq_1 \in Seq_1$ we know that there exists a $w_1 \in L(D)$ such

that $seq_1 \equiv w_1$, and there also exists a $seq_2 \in Seq_2$ such that $seq_2 \equiv w_1$. Even if seq_1 and seq_2 are different, they represent the same order of activities, and therefore construct the same behavior in M^{Seq}. An example of different dispatching sequences that construct the same behavior is: $seq_1 = A1; A2; (A1; A2)^\infty$ and $seq_2 = A1; (A2; A1)^\infty$.

H. Continuous behavior

In the above parts automata are constructed that model the behavior of an LSAT specification. In the conversion from LSAT to the automata, some information was not translated, such as timing and coordinates of the positions of the peripherals. In this section we shortly discuss how this information can be contained in the automata model.

Hybrid automata [17] are automata with a set of continuous variables whose values are defined by differential equations. The differential equation for a continuous variable can be dependent on the state of the automaton, and the continuous variables can be updated (can jump value) when an event occurs. The peripheral automata described in Section III-F can be modified to hybrid automata with intermediate states that represent an action actively taking place. A transition leaving these intermediate states can only occur if a certain amount of time has passed, or the target location has been reached. The time and position are modeled as continuous variables that evolve by a differential equation. CIF has the functionality to model such hybrid automata.

IV. CONCLUSIONS

LSAT is a model-based engineering tool used for concept flexible manufacturing system design and analysis. Using a domain specific language, a system can be specified in LSAT. LSAT has the functionality of analysis, synthesis, visualization and validation that enables to quickly pinpoint improvement areas of the system specification. An LSAT specification consists out of peripherals, resources, actions, activities and dispatching sequences. In this paper, the semantics and formalisms of such a specification are elaborated upon. An action-level behavioral model using automata is given for a system specified in LSAT. For some LSAT specifications, this behavioral description may require an infinite amount of states and transitions. This infinite explicit action-level model can act as a baseline in future work, where it can be investigated if verification and synthesis perhaps can be applied to a finite abstraction of this statespace, for action-level behavioral guarantees.

REFERENCES

[1] T. Basten, J. Bastos, R. Medina, B. van der Sanden, M. Geilen, D. Goswami, M. Reniers, S. Stuijk, and J. Voeten, *Scenarios in the Design of Flexible Manufacturing Systems*. Cham: Springer, 2020, pp. 181–224.

[2] S. Adyanthaya, H. Ara, J. Bastos, A. Behrouzian, R. Sánchez, J. van Pinxten, B. van der Sanden, U. Waqas, T. Basten, H. Corporaal, R. Frijns, M. Geilen, D. Goswami, M. Hendriks, S. Stuijk, M. Reniers, and J. Voeten, "xCPS: A tool to explore cyber physical systems," *SIGBED Rev.*, vol. 14, no. 1, p. 81–95, Jan. 2017.

[3] B. van der Sanden, M. Reniers, M. Geilen, T. Basten, J. Jacobs, J. Voeten, and R. Schiffelers, "Modular model-based supervisory controller design for wafer logistics in lithography machines," in *ACM/IEEE Int. Conf. Model Driven Eng. Lang. Syst.*, 2015, pp. 416–425.

[4] B. van der Sanden, J. Bastos, J. Voeten, M. Geilen, M. Reniers, T. Basten, J. Jacobs, and R. Schiffelers, "Compositional specification of functionality and timing of manufacturing systems," in *Forum Specification Des. Lang.*, 2016, pp. 1–8.

[5] C. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems*. Springer, 2010.

[6] F. Baccelli, G. Olsder, J.-P. Quadrat, and G. Cohen, *Synchronization and linearity. An algebra for discrete event systems*. Chichester: Wiley, 1992.

[7] A. Dasdan, "Experimental analysis of the fastest optimum cycle ratio and mean algorithms," *ACM Trans. Design Autom. Electr. Syst.*, vol. 9, pp. 385–418, Oct 2004.

[8] M. Geilen and S. Stuijk, "Worst-case performance analysis of synchronous dataflow scenarios," in *IEEE Proc. Int Conf. Hardware/Softw. Codesign Syst. Synthesis*. Assoc. Computing Machinery, 2010, p. 125–134.

[9] B. van der Sanden, M. Geilen, M. Reniers, and T. Basten, "Partial-order reduction for performance analysis of max-plus timed systems," in *Int. Conf. Appl. Concurrency Syst. Des.*, 2018, pp. 40–49.

[10] J. Kelley and M. Walker, "Critical-path planning and scheduling," in *Eastern Joint IRE-AIEE-ACM Comput. Conf.* Assoc. Comput. Machinery, 1959, p. 160–173.

[11] D. van Beek, W. Fokkink, D. Hendriks, A. Hofkamp, J. Markovski, J. van de Mortel-Fronczak, and M. Reniers, "CIF 3: Model-based engineering of supervisory controllers," ser. LNCS, vol. 8413, 2014, pp. 575–580.

[12] B. van der Sanden, "Performance analysis and optimization of supervisory controllers," Ph.D. dissertation, Dept. Elec. Eng., Eindhoven Univ. of Tech., Nov. 2018.

[13] D. Vose, *Risk Analysis: A Quantitative Guide*. Wiley, 2008.

[14] P. Lambrechts, "Trajectory planning and feedforward design for electromechanical motion systems. DCT rep. 2003.008," 2003.

[15] A. Ghamarian, M. Geilen, S. Stuijk, T. Basten, B. Theelen, M. Mousavi, A. Moonen, and M. Bekooij, "Throughput analysis of synchronous data flow graphs," in *Int. Conf. Appl. Concurrency Syst. Des.*, 2006, pp. 25–36.

[16] C. Seatzu, M. Silva, and J. Van Schuppen, *Control of discrete-event systems*, ser. LNCIS. Springer, 2013, vol. 433.

[17] R. Alur, C. Courcoubetis, T. Henzinger, and P. Ho, "Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems," in *Hybrid Syst.* Berlin, Heidelberg: Springer, 1993, pp. 209–229.