ORIGINAL ARTICLE

Journal Section

Time- and Behavior-Preserving Execution of Determinate Supervisory Control

Alireza Mohammadkhani[1*] | **Marc Geilen**[1*] | **Jeroen Voeten**[1*] | **Twan Basten**[1*]

[1]Department of Electrical Engineering, Eindhoven University of Technology, Eindhoven, The Netherlands

Correspondence
Alireza Mohammadkhani, Department of Electrical Engineering, Eindhoven University of Technology, PO Box 513, 5600 MB, Eindhoven, The Netherlands
Email: a.mohammadkhani@tue.nl

Present address
[*]Department of Electrical Engineering, Eindhoven University of Technology, PO Box 513, 5600 MB, Eindhoven, The Netherlands

Funding information
EU ECSEL Joint Undertaking, Grant Number 826452 (project Arrowhead Tools)

Behaviors and activities are natural concepts (found, e.g., in UML and SysML) for model-driven design of Cyber-Physical Systems (CPS). These concepts are formalized in the *activity framework*, a model-based framework incorporating a model of activities with determinate timing and behavior, and a strong mathematical foundation based on max-plus algebra that allows efficient timing analysis and optimization. It provides a layered view of the system's actions and events, activities built from them, and sequences of activities that capture the overall behavior of the system. Implementations of supervisory control for CPS to govern the system behavior are often made by hand. Preserving the specified behavior and the model-predicted timing in an implementation is challenging, due to the need to simultaneously handle *action timing*, *synchronization*, *concurrency*, *pipelining*, and *plant feedback*. We introduce an execution architecture and engine to automatically synthesize an implementation of a supervisory controller directly from a model specification. The execution engine is guaranteed to execute a specification in a time- and behavior-preserving fashion, even in the presence of action timing variations and including event feedback in a physical execution. We prove that the architecture and engine preserve the specified ordering of actions and events of the model as well as

the timing thereof, up to a known bound. We validate our approach on a prototype production system.

KEYWORDS
cyber-physical systems, model-driven design, time- and behavior-preserving execution, discrete event systems, supervisory control, synthesis

1 | INTRODUCTION

Modeling formalisms such as SysML [1] and UML [2] are widely used to develop systems and model their behaviors [3], including specification and analysis of Cyber-Physical Systems (CPS) [4, 5]. The behaviors of CPS are typically specified using the *activity* views provided by SysML and UML. Flexible Manufacturing Systems (FMS) are a prime example of CPS, in which integrated, computer-controlled, complex of automated material handling devices process products with reduced overhead time, effort, and cost to change production line configurations [6].

To properly implement the desired behaviors of such CPS, engineers face challenges, mainly due to the integral control of *timing*, *synchronization*, *concurrency*, and *pipelining* of actions and activities, while properly and timely responding to *event feedback from the plant*. Handling all these aspects simultaneously, while adhering to the specification models, is difficult using current engineering methods. The combination of these aspects in a physical execution may easily result in non-deterministic behavior that is inconsistent with the intended specification. This makes it extremely difficult to qualify the system.

The activity framework [7] is an activity-based formalism that focuses on *determinate* activity models that include activity timing and behavior. It has a strong mathematical foundation, based on max-plus algebra, that allows for efficient and scalable timing analysis and optimization. It supports a modular and scalable model-based design methodology for FMS that focuses on the *supervisory control* layer that controls the system by orchestrating the actions and events that the *plant* layer executes. The framework is being used by various industrial domains (such as lithography, chip packaging, production printing and warehousing) for specification, mechanical layout studies, and performance analysis of FMS [8, 9]. It provides an intuitive specification language that abstracts system actions, plant events and their dependencies into activities. Each such activity represents a *determinate* piece of system behavior. As an example, consider the activity of turning a part during production with a gripper, consisting of gripping, movement, and rotation actions; initial movement precedes gripping, which in turn precedes further movement with rotation occurring concurrently. Determinacy in the context of the activity framework means that variations in timing or feedback within an activity do not influence the behavior of the system for that activity. The activity abstraction provides timing analysis techniques rooted in max-plus linear systems theory. The framework uses a modular approach to logistics controller specification based on automata theory. It also provides scalable best-case and worst-case makespan and throughput analysis through analysis of max-plus linear systems.

The activity framework provides model-based design [10] support for FMS. The behavior of the system is modeled and analyzed to obtain a design that satisfies the system requirements. It helps the design process by providing models that are understandable by engineer and amenable to mathematical analysis and to synthesis. The framework currently requires manual implementation of the supervisory control layer after its design is finalized, which is cumbersome and error-prone, particularly considering aspects of timing and concurrency. It is difficult to make sure that the realization conforms to the specification. This article provides a solution to replace this manual process with

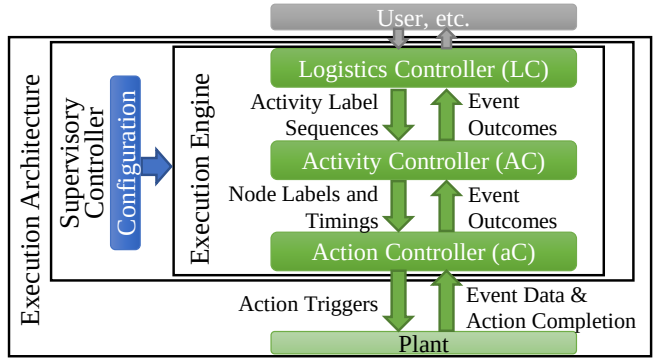


FIGURE 1 The execution architecture

a model-driven synthesis [11] approach, where models are used to drive the implementation process. This has the potential to reduce the cost and time required to deploy these systems. It also significantly reduces the errors in the implementation process, and yields a system that preserves the prescribed model behavior and satisfies timing and performance requirements, by construction.

In this article, we introduce an execution architecture and an execution engine, called the *Activity Execution Engine*, to implement supervisory control of FMS automatically from specifications of activity models that include *event feedback* and responses to such feedback. This provides a mechanism to respond to feedback from the plant layer of an FMS in the specification of the supervisory controller. The handling and timing of behaviors with event feedback can be captured as a switched max-plus linear system [12]. We define the operational characterization of the execution in this article. We consider *Events* as planned notifications of outcomes of actions from the plant to the supervisory controller that enable the controller to react to occurrences of interest within the plant. Events have a finite set of possible *outcomes* that are non-deterministically resolved at runtime. Any physical execution will have timing deviations, which may affect the timing and ordering of other actions and events, admitting undesired non-deterministic behavior. This makes it difficult to ensure that the system behavior is consistent with the specified behavior in the model and that its desirable (timing) properties are preserved. Our execution architecture and engine guarantee that any behavior that is executed respects the occurrence, the order, and the timing of actions and activities as prescribed by the model. By preserving timing and behavior, our execution engine also guarantees any performance predictions obtained from the model.

The execution architecture that we present in this article is illustrated in Figure 1. It consists of the physical plant and our proposed execution engine, which is the implementation of the supervisory controller. The execution engine is configured with an activity model (with event feedback) of an FMS as input. It executes the model on the plant. It is organized in three sublayers, responsible for logistics, activity execution, and action execution, respectively, with well-defined interfaces.

We formulate the correctness relation between the behavior specified by the model and the physical execution that captures the preservation of the ordering of actions and events of a behavior and their timing, up to a specified bound, respecting the outcomes of events received during the execution. We then proceed to prove that our execution engine is time- and behavior-preserving by showing that the relation holds for our implementation. The actions and events that are executed only depend on the specification and the outcomes of events that materialize at run-time and not on any timing deviation of actions or events due to the specifics of hardware and/or software implementation in the execution. Thus, the behavior of the system is determinate up to the resolution of the outcomes of events. We

Abbreviation	
CPS	Cyber-Physical Systems
FMS	Flexible Manufacturing Systems
DAG	Directed Acyclic Graph
LC	Logistics Controller
AC	Activity Controller
aC	Action Controller
xCPS	eXplore Cyber-Physical Systems
SC	Supervisory Control
FSM-SADF	Finite State Machine-based Scenario-Aware Data-Flow

TABLE 1 Table of abbreviations

validate our solution on a prototype production system. Although we use the activity framework and FMS as a driver for our work, we believe that our approach can be adapted to controller synthesis in other modeling approaches focusing on determinate behavior and to other types of CPS.

Contributions

The main contributions of this article are as follows:

- a layered, model-driven supervisory-control execution architecture (Figure 1) with well-defined interfaces between the layers;
- an activity execution engine that takes a supervisory controller model, i.e., an activity model with event feedback, as input and executes this model in a determinate fashion, up to a specified bound on the timing of actions and the run-time resolution of events;
- a precise characterization of time- and behavior preservation and a proof that our execution architecture and engine preserve timing and behavior;
- a validation of the framework on a prototype production system.

The remainder of this article is organized as follows. In Section 2, we examine related literature and compare the state of the art to our approach. In Section 3, we describe the modeling framework and present a model of a simple running example. In Section 4, we discuss the timing and behavior relation that our execution architecture preserves between the model and the execution. Section 5 precisely defines the problem that we address in this article. Section 6 describes the plant model that is assumed in the architecture, as well as the execution engine, and proves that the execution engine preserves timing and behavior of the specification. We validate our proposed execution architecture and execution engine in Section 7. Finally, Section 8 concludes our work. Tables 1 and 2 provide overviews of all abbreviations and notations used throughout the article.

Notation	
$\mathbb{A}, \mathbb{S}, \mathbb{P}, \mathbb{E}, \mathbb{O}$	sets of actions, resources, peripherals, events, event outcomes (in an activity model)
$\mathcal{R}(p)$	the resource that contains peripheral p
$Nod, \rightarrow$	sets of nodes (Nod) and dependencies ($\rightarrow$) in an activity
$\mathcal{M}, \mathcal{T}$	mapping ($\mathcal{M} : Nod \rightarrow \mathbb{A} \times \mathbb{P} \cup \mathbb{S} \times \{rI, cI\} \cup \mathbb{E}$) of nodes to types and timing ($\mathcal{T} : Nod \rightarrow \mathbb{R}_{\geq 0}$) of nodes in an activity
$Pred(n)$	the set of predecessor nodes of n
Q, S, Σ, F, Ω	sets of states, initial states, labels, final states, and the transition relation of an I/O automaton
$Acts$	a finite set of activities
γ	relation between events and their possible outcomes
Y	an I/O automaton with (event, outcome) pairs as input labels and activities as output labels
λ	the empty input label denoting a transition that has no (event, outcome) pair as input
ϵ	the empty activity without actions or events
$\lvert w \rvert^e$	the number of times event e is processed in word w
τ	empty sequence of activities
$\mathcal{B}$	the behavior activity function (that converts a behavior into a single activity)
$Act^P = \mathcal{B}(P)$	the activity of behavior P
$\mathcal{V}$	the processed events function (that provides the sequence of events processed in a behavior)
$U^P = \mathcal{V}(P)$	sequence of (event, outcome) pairs corresponding to behavior P
$\bar{x}$	resource availability function (that provides the availability time for each resource)
$S(n)$, $S(n, \bar{x})$	the specified model start time of node $n \in Nod$ (given resource availability $\bar{x}$)
$C(n)$, $C(n, \bar{x})$	the specified model completion time of node $n \in Nod$ (given resource availability $\bar{x}$)
S^P, C^P	specified model start and completion times of nodes in behavior P
$S'(n), C'(n)$	the start and completion time of node $n \in Nod$ in a physical execution
$\mathcal{T}'(n)$	event delay of event node n with $\mathcal{M}(n) \in \mathbb{E}$
Ψ	the start time of a physical execution
$\delta(n)$	execution delay of node n, the delay between triggering an action and its actual execution
$\mathbb{D}_{\mathbb{A}}$	upper bound on the execution delays $\delta(n)$ of all nodes n
$\mathbb{D}_{\mathbb{E}}$	the maximum time required to process events
ψ	wall clock time of the action controller (aC)
$S(\rho)$, $S(\rho, \bar{x})$	the specified model start time of decision path ρ (given resource availability $\bar{x}$)
S^ρ, C^ρ	model start and completion times of action nodes in ρ
$\mathbb{D}_{Event}$	maximum event outcome sending time from action controller (aC) to logistics controller (LC)
$\mathbb{D}_{LC}, \mathbb{D}_{AC}, \mathbb{D}_{aC}$	the maximum time that the LC, AC, and aC take to process an event outcome, respectively
$S(A)$, $S(A, \bar{x})$	the specified start time of activity A (given resource availability $\bar{x}$)
$S(\rho)$, $S(\rho, \bar{x})$	the specified start time of decision path ρ (given resource availability $\bar{x}$)

TABLE 2 Table of notations

2 | RELATED WORK

In this section, we discuss the literature relevant to our work. We present a way to synthesize a model-driven implementation given a set of assumptions on the plant and the model. There exist two main types of approaches to implementation synthesis. One set of approaches aims to configure an engine that is designed specifically for a given model of computation [13, 14, 15, 16] and other approaches rely on code generation [17, 18]. Our work fits within the former category. Implementation synthesis approaches may preserve a variety of properties, such as behavior and timing. Our work preserves timing as well as behavior, where behavior is preserved through timing guarantees and respecting event outcomes.

Similar to our approach, where the specification is used to configure an execution engine, an actor-based model is used to configure an execution run-time component for distributed real-time embedded systems in the PTIDES framework [13, 14]. Every actor has a set of input ports and output ports and a function that captures the timing relation between them. The framework provides a mechanism to decide whether it is safe to process an input event at any input port or not, in a distributed execution setting. An event is considered safe to process if it can be guaranteed that no other event will be arriving in the future that invalidates the current event in light of the model specification. This is achieved by enriching real-time indeterminate implementation events with deterministic logical timestamps, decoupled from such physical timing variations. Combined with analysis of the real-time properties of the implementation, relations can be determined and guaranteed between the implementation and the model. This method ensures that the execution is in accordance with the model, independent from non-deterministic physical (distributed) timing variations. Specifically, if the model is determinate, then so is the implementation. Activities can be mapped to the actor-based model of [13, 14]. However, such a mapping would be at the expense of the scalability and analysis advantages of our model that come from the multiple levels of abstraction for specification (actions, activities, automata).

The Sparse Synchronous Model [19] is an imperative programming language, supporting dynamic process creation, recursion, and run-time scheduling. It is targeting real-time embedded software, allowing programmers to precisely specify timing aspects as part of the program. SSM decouples specified timing from platform-specific physical timing, via a platform-specific runtime environment. SSM is inspired by PTIDES and synchronous languages [20], combining concepts from event-driven execution as in PTIDES with the determinism that is inherent in the semantic and execution model underlying the family of synchronous languages. It provides flexibility while ensuring that the execution adheres to the specified timing. It trades expressiveness and flexibility for analyzability. In contrast, our approach is targeting model-driven supervisory control of CPS, instead of embedded software, where the model supports efficient and scalable analysis.

Another example of configuring an execution engine with a specification can be found in [15], where Finite State Machine-based Scenario-Aware Data-Flow (FSM-SADF) [21] is used to specify a multi-processor system. The execution architecture is configured by allocating resources to actors. Since the approach uses time-division multiplexing, these resources include processor cores and memory, as well as time slots of those cores to guarantee timing. The number of scenarios to be executed affects resource budgets and switching time. Pipelining of multiple scenarios is possible and scenarios are executed according to a sequence. The execution of scenarios is deterministic and data-driven which allows a predictable timing and worst-case guarantees for them. Our approach also provides worst-case behavior timing guarantees assuming worst-case timing for actions, but our execution engine follows a time-driven execution following model-prescribed system timing instead of considering the model timing as a worst-case upper bound only.

Petri nets [22] are a well-known formalism that could be used to automatically configure an execution engine. An

example can be found in [16], where they are directly executed on an FMS. A Petri net is systematically specified such that properties such as liveness and boundedness are guaranteed. A supervisory controller is then given from this Petri net that preserves these properties. Similar to our architecture, this is executed on the controller of an FMS that executes the individual actions of the system. The controller runs a Petri net execution engine that is configured with the specification of the supervisory controller. The order of actions is preserved by the execution engine while timing of actions is not considered. We assume timed actions and guarantee timing and behavior preservation between the model and the execution. Furthermore, we enable the supervisory controller to make decisions at runtime based on different outcomes of events, which is not possible in [16].

Next, we consider code generation methods such as [18], where timed automata [23] are used for code generation for real-time systems. The model is extended with task graphs to describe dependencies between tasks. Resources of a task are mapped to semaphores that are used to ensure resource use according to the specification. The automaton has guards that have clock (i.e., time) and variable constraints. Similar to our work, a deterministic automaton is used. Determinism is ensured by enforcing priorities to action transitions that are simultaneously enabled and could cause non-deterministic behavior in the automaton. Timing assumptions are made for the plant that are similar to ours, namely Assumptions 1 and 2 introduced in Section 6.1. The three levels of abstraction in our framework for specification (actions, activities, automata) provide more scalability in specification and analysis (using max-plus linear matrix-vector multiplications) as shown in [7].

Another code-generation method, which is considered a follow-up of the PTIDES approach [13, 14], is underlying the Lingua Franca line of work [17, 24, 25]. Lingua Franca focuses on providing a deterministic execution for concurrent actors (called *reactors* in the method) that react to incoming events carrying logical time stamps. Communication between reactors creates dependencies that are similar to our DAG based specifications. The framework supports the automatic generation of the coordination code that ensures correct execution for a number of different programming languages. Every reactor is then mapped to a block of generated code with guarantees that concurrent reactors are executed in a deterministic manner, meaning that a set of inputs always gives a predictable set of outputs. Similar to behavior preservation in our approach, perturbations in the timing of reactors do not change the behavior of the system. Recent work details the adding and removing of reactors at runtime [24] and layered scheduling, which decouples actors in layers that may be specified with their own deadlines [25]. As for the PTIDES approach of [13, 14], activities can be mapped to the Lingua Franca model, with the advantage of giving a deterministic execution as opposed to the PTIDES execution model. The same scalability advantages (concerning specification and analysis) of our framework over PTIDES hold for Lingua Franca.

Our action controller realizes time-triggered execution similar to the time-triggered architecture introduced in [26], where real time is modeled with instances of time with fixed intervals in between them, called *durations*. A duration can be of activity (when the modeled system is active, not to be confused with the term activity in this article) or of silence (when it is not). In our approach, time is not divided into fixed intervals. The time instances of our model are derived from the specification timings. Our execution engine follows these timings and does not incorporate a predefined set of fixed durations where it is active or silent. Our action controller is triggered by the start and completion times of the nodes that it executes and event information that it receives.

The logistics controller of our execution engine is configured with a deterministic I/O automaton. We do not make any assumptions on how the logistics automaton is obtained. The I/O automaton could be synthesized using the Supervisory Control theory of [27]. In such an approach, the activities would be considered *controllable* actions that may emit events, and the possible outcomes of those events would be considered *uncontrollable* actions which we respond to by a sequence of activities. Such an approach would guarantee desirable properties such as non-blockingness.

3 | SYSTEM SPECIFICATION, BEHAVIOR AND TIMING

In this section, we explain the activity framework [7], including its extension with event-feedback, as introduced in [12]. We further explain how it captures system behavior and timing, which are preserved by our execution engine. This section is organized into three subsections, corresponding to specification, behavior, and timing, respectively.

3.1 | System Specification

A system model consists of the following elements, explained in more detail below: a set of activities, an I/O automaton that specifies the logistics controller as part of the supervisory controller that captures the possible sequences of activities that can be executed, and a set of events and their possible outcomes. Activities are composed of actions and events and the dependencies (precedence relations) between them. The I/O automaton captures the response of the logistics controller to the outcomes of events in terms of the activity sequences that it executes.

The basis of an activity model is a set of *resources*, for instance a turner robot within an FMS that orients pieces correctly. Such a resource can have *peripherals* that perform *actions*. For example, it can have a motor to perform the action to pick up a workpiece, another motor that can rotate a workpiece, and a gripper that can grab or release a workpiece. The supervisory control of a machine also requires feedback from the machine. For example, if a turner fails to pick up a workpiece, the controller needs to respond accordingly. For this purpose, the activity framework incorporates event feedback [12]. We define the corresponding operational characterization of activity models including event feedback in this article. *Events* are anticipated notifications of outcomes of activities from the plant to the supervisory controller, which reacts to them. They are non-deterministic (from the model perspective) resolved at runtime. The turner, for instance, may have an event reporting success or failure, every time it attempts to pick up a workpiece. To turn a workpiece, the *turn* activity is performed where the turner moves and grabs a workpiece, moves to a position where it can safely turn, turns the workpiece, moves back to the position where it grabbed the workpiece, and finally releases the workpiece in a determinate sequence of actions. Such a turn station may have a sensor that indicates whether the turner was able to pick up the workpiece or not and report the outcome of the event that the activity turn emits with one of two possible outcomes, *success* or *failure*.

The key concept in the activity framework are activities. We first informally introduce activities using the example of Figure 2, and then we formally define them in Definition 1. The activity framework models activities as Directed Acyclic Graphs (DAGs) [28], which are a common way of modeling the behavior of CPS [29, 17]. In an activity DAG, nodes are either *actions* performed on peripherals, *claims* or *releases* of a *resource*, or *events* being emitted by the activity. The edges represent precedence constraints between them. The concept of resource allows the model to impose ordering constraints on actions across concurrent, pipelined activities to ensure correctness of behavior and validity of execution. Resources have to be *claimed* and *released* to perform actions on the peripherals of those resources.

To illustrate the concepts of an activity model, we introduce a running example and use it throughout this section. Figure 2 depicts activities Act_1, Act_2, Act_3, and Act_4 of the running example. Action nodes are colored in the figure and labeled (for example n_1 in Act_1) and their timing durations are shown between parentheses (for instance, duration 1 for node n_1). We omit node identifiers for resource claim or release nodes for brevity. For example, the three leftmost nodes in Act_1 correspond to resource claims, indicated with small colorless circles without identifier. We use labels on top or at the bottom of each node to depict whether they are actions on peripherals, events, or claims or releases of resources. For example, node n_1 represents action a being performed on peripheral p_1 and the top left node in Act_1 corresponds to a claim on resource r_1. To avoid clutter in Act_1, resources appear out of order in that activity.

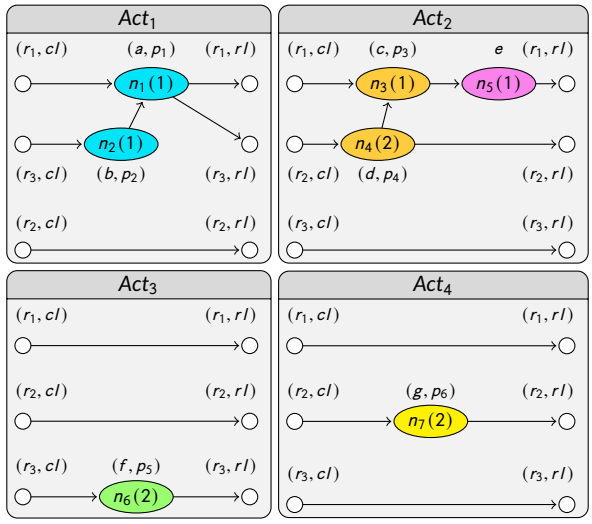


FIGURE 2 Activities Act_1, Act_2, Act_3, and Act_4 of the running example. Colors are used to distinguish between the action nodes of different activities and also between action and event nodes.

Arrows denote dependencies, i.e., the edges of the DAG. For example, in Act_1, n_2 has to complete before n_1 can start. Event nodes, such as n_5 in Act_2 with event e, represent notifications from the plant layer to the supervisory control layer. Each event has one or more known possible outcomes and depending on the outcome that is received, the logistics controller responds by executing a different sequence of activities. It is convenient to have all activities claim and release all resources, as illustrated in the four activities of Figure 2, even if a resource is not actually used (as e.g. resource r_2 in Act_1). Resource claims always precede actions on peripherals of a claimed resource. Claimed resources are always released again. These conditions, and some others formalized in Definition 1 below, ensure the soundness of activity definitions.

The activity framework provides a layered model with different levels of abstraction, from individual actions, to activities, and sequences of activities, captured by a finite state automaton. The automaton uses activities on its transition labels, making it significantly smaller than a logistics controller that would be specified in terms of individual actions, which enhances scalability.

The activity framework formalizes these concepts as follows. Let $\mathbb{A}$ denote the set of actions, let $\mathbb{S}$ denote the set of resources, and let $\mathbb{P}$ denote the set of peripherals. We assume a function $\mathcal{R} : \mathbb{P} \to \mathbb{S}$, such that $\mathcal{R}(p)$ is the resource that contains peripheral p. To model event feedback, we further let $\mathbb{E}$ be the set of events and $\mathbb{O}$ be the set of possible event outcomes (of all events). We define an activity as follows.

Definition 1 *(Activity). An activity is a quadruple* $(Nod, \to, \mathcal{M}, \mathcal{T})$ *where*

- $(Nod, \to)$ *is a DAG, with* Nod *a set of nodes and* $\to \subseteq Nod \times Nod$ *a set of dependencies;*
- $\mathcal{M} : Nod \to \mathbb{A} \times \mathbb{P} \cup \mathbb{S} \times \{rl, cl\} \cup \mathbb{E}$ *is a function that maps every node to an action label* (a, p) *denoting action* a *on peripheral* p*, a release or claim label where* (r, rl) *denotes a release of resource* r *and* (r, cl) *denotes a claim of resource* r*, or an event;*
- $\mathcal{T} : Nod \to \mathbb{R}_{\geq 0}$ *is a function that assigns a non-negative time value to every node such that* $\mathcal{T}(n) = 0$ *if* $\mathcal{M}(n) \in \mathbb{S} \times \{rl, cl\}$*;* $\mathcal{T}$ *denotes* specified duration *of an action if* $\mathcal{M}(n) \in \mathbb{A} \times \mathbb{P}$*, and it denotes* specified delay *of an event*

if $\mathcal{M}(n) \in \mathbb{E}$.

We write $n_1 \rightarrow n_2$ *to denote that* $(n_1, n_2) \in \rightarrow$. *We denote the transitive closure of* $\rightarrow$ *by* $\rightarrow^+$. *We define the predecessor nodes of n as:*

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

An activity satisfies the following constraints:

I. All nodes mapped to the same peripheral are sequentially ordered to avoid self-concurrency on a single peripheral:
$\forall_{n_1,n_2 \in Nod, a_1,a_2 \in \mathbb{A}, p \in \mathbb{P}}\ n_1 \neq n_2 \wedge \mathcal{M}(n_1) = (a_1, p) \wedge \mathcal{M}(n_2) = (a_2, p) \Rightarrow n_1 \rightarrow^+ n_2 \vee n_2 \rightarrow^+ n_1$

II. Each resource is claimed exactly once:
$\forall_{r \in \mathbb{S}}\ \exists_{n_1 \in Nod}\ \mathcal{M}(n_1) = (r, cl) \wedge \nexists_{n_2 \in Nod}\ n_1 \neq n_2 \wedge \mathcal{M}(n_2) = (r, cl)$

III. Each resource is released exactly once:
$\forall_{r \in \mathbb{S}}\ \exists_{n_1 \in Nod}\ \mathcal{M}(n_1) = (r, rl) \wedge \nexists_{n_2 \in Nod}\ n_1 \neq n_2 \wedge \mathcal{M}(n_2) = (r, rl)$

IV. Every action node is preceded by a claim node on the corresponding resource:
$\forall_{n_1 \in Nod, a \in \mathbb{A}, p \in \mathbb{P}, r \in \mathbb{S}}\ \mathcal{M}(n_1) = (a, p) \wedge \mathcal{R}(p) = r \Rightarrow \exists_{n_2 \in Nod}\ \mathcal{M}(n_2) = (r, cl) \wedge n_2 \rightarrow^+ n_1$

V. Every action node is succeeded by a release node on the corresponding resource:
$\forall_{n_1 \in Nod, a \in \mathbb{A}, p \in \mathbb{P}, r \in \mathbb{S}}\ \mathcal{M}(n_1) = (a, p) \wedge \mathcal{R}(p) = r \Rightarrow \exists_{n_2 \in Nod}\ \mathcal{M}(n_2) = (r, rl) \wedge n_1 \rightarrow^+ n_2$

VI. Every release node is preceded by a claim node on the corresponding resource:
$\forall_{n_2 \in Nod, r \in \mathbb{S}}\ \mathcal{M}(n_2) = (r, rl) \Rightarrow \exists_{n_1 \in Nod}\ \mathcal{M}(n_1) = (r, cl) \wedge n_1 \rightarrow^+ n_2$

VII. Every claim node is succeeded by a release node on the corresponding resource:
$\forall_{n_1 \in Nod, r \in \mathbb{S}}\ \mathcal{M}(n_1) = (r, cl) \Rightarrow \exists_{n_2 \in Nod}\ \mathcal{M}(n_2) = (r, rl) \wedge n_1 \rightarrow^+ n_2$

VIII. No node precedes a claim node:
$\forall_{n_1 \in Nod, r \in \mathbb{S}}\ \mathcal{M}(n_1) = (r, cl) \Rightarrow \nexists_{n_2 \in Nod}\ n_2 \rightarrow^+ n_1$

IX. Release nodes are not succeeded by any node:
$\forall_{n_1 \in Nod, r \in \mathbb{S}}\ \mathcal{M}(n_1) = (r, rl) \Rightarrow \nexists_{n_2 \in Nod}\ n_1 \rightarrow^+ n_2$

X. Every event node is preceded by another node:
$\forall_{n_1 \in Nod}\ \mathcal{M}(n_1) \in \mathbb{E} \Rightarrow \exists_{n_2 \in Nod}\ n_2 \rightarrow^+ n_1$

XI. Multiple instances of the same event in an activity must be sequentially ordered:
$\forall_{n_1,n_2 \in Nod}\ \mathcal{M}(n_1), \mathcal{M}(n_2) \in \mathbb{E} \wedge \mathcal{M}(n_1) = \mathcal{M}(n_2) \Rightarrow n_1 \rightarrow^+ n_2 \vee n_2 \rightarrow^+ n_1$

We denote the universe of activities with $\mathbb{ACT}$.

Nodes that are mapped to an action label are called *action nodes*. Nodes that are mapped to a release or claim label are called *resource nodes*, and nodes that are mapped to an event are called *event nodes*.

The constraints in the above definition make sure that an activity captures deterministic behavior that can be executed without erroneous behavior such as deadlocks, actions happening without their resource being available,

resources not being released after an activity no longer needs them, etc. It is not difficult to see that the four activities in Figure 2 satisfy all constraints. Constraints (VIII-XI) extend the existing constraints of the activity framework [7]. Also constraints (II) and (III) are strengthened. We explain why (II), (III), (VIII), and (IX) are needed when we discuss sequencing of activities. Constraint (X) requires that event nodes are preceded by at least one node so that they depend on at least one resource. This is because the event has to originate from some component of the system and the framework uses the concept of resources to model system components. Constraint (XI) requires that multiple instances of the same event in an activity are sequentially ordered. This ordering is necessary to maintain a deterministic behavior when we process the outcomes of the event instances.

We use an I/O automaton to specify the logistics controller. An I/O automaton is defined as follows.

Definition 2 *(I/O Automaton). An I/O automaton [30] is a seven-tuple $(Q, S, \Sigma, I, O, \Omega, F)$ where*

- *Q is a finite set of states;*
- *$S \subseteq Q$ is a set of initial states;*
- *Σ is a finite set of labels and I and O are disjoint sets of input and output labels, respectively, such that $I \cup O = \Sigma$;*
- *$\Omega \subseteq Q \times I \times O \times Q$ is a transition relation;*
- *$F \subseteq Q$ is a set of final states.*

We write $q_1 \rightarrowtail q_2$ if $(q_1, i, o, q_2) \in \Omega$ for some i and o, and denote the transitive closure of $\rightarrowtail$ by $\rightarrowtail^+$.

With the ingredients of the activity framework defined, we formally define an activity specification as follows.

Definition 3 *(Activity Specification). An activity specification is a five-tuple (Acts, $\mathbb{E}$, $\mathbb{O}$, γ, Y) where*

- *Acts is a finite set of activities;*
- *$\mathbb{E}$ is a finite set of events including all events from the activities in Acts;*
- *$\mathbb{O}$ is a finite set of event outcomes;*
- *$\gamma \subseteq \mathbb{E} \times \mathbb{O}$ specifies the relation between events and their possible outcomes;*
- *Y is an I/O automaton with pairs of events and their outcomes as input labels and activities as output labels, i.e., $I = \mathbb{E} \times \mathbb{O} \cup \{\lambda\}$ and $O = Acts \cup \{\epsilon\}$ where λ is the empty input label denoting a transition that has no event and outcome pair, and ϵ is the empty activity without actions or events.*

With the ingredients of an activity model defined, we model the running example as follows.

- $\mathbb{A} = \{a, b, c, d, f, g\}$
- $\mathbb{P} = \{p_1, p_2, p_3, p_4, p_5, p_6\}$
- $\mathbb{S} = \{r_1, r_2, r_3\}$
- $Acts = \{Act_1, Act_2, Act_3, Act_4\}$ as given in Figure 2
- $\mathbb{E} = \{e\}$
- $\mathbb{O} = \{u_1, u_2\}$
- $\gamma = \{(e, u_1), (e, u_2)\}$
- I/O automaton Y depicted in Figure 3

The system modeled in the running example has six actions (in $\mathbb{A}$), each executed on a specific peripheral (in $\mathbb{P}$) of one of the three resources (in $\mathbb{S}$). The mapping from actions to peripherals and resources follows from the activity

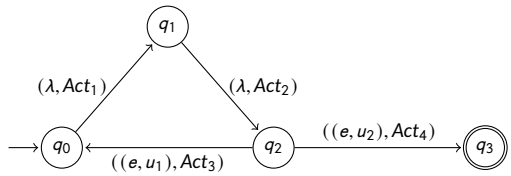


FIGURE 3 I/O automaton (Y) of the running example

definitions in Figure 2. The system plant emits one event e with two possible outcomes u_1 or u_2 as specified by γ. The delay of event e is specified in the DAG of activity Act_2 in Figure 2, which shows that event e occurs one time unit after the completion of n_3. The I/O automaton specifying the logistics controller is shown in Figure 3. It has four states, with q_0 the initial state and q_3 the only final state. The transitions are labeled with ((event, event outcome), activity) tuples, where the (event, event outcome) pair may be the empty input λ. The controller starts from state q_0 and first executes Act_1 and Act_2. Then, reaching state q_2, based on the outcome of event e, which is emitted by node n_5 of activity Act_2 (as seen in Figure 2), it executes either Act_4 in response to outcome u_2 and goes to final state q_3, or Act_3 in response to outcome u_1 and returns to q_0. This shows that, for instance, the sequence of three activities Act_1 followed by Act_2 followed by Act_4 is an acceptable behavior of the modeled system, since q_3 is a final state. The precise semantics of an activity specification, i.e., the (acceptable) behaviors that it specifies and their timings, is formalized in the next two subsections.

For proper behaviors of a controller, we require that an automaton in an activity specification does not process an event that has not been emitted yet by an activity. We further require that the automaton has processed all events that are emitted before reaching a final state. This ensures that the number of events emitted in the system cannot accumulate without limit. An I/O automaton is *consistent* (Definition 18 of [12]) if it follows these two assumptions and does not contain states that do not lead to final states.

We further assume the I/O automaton specifying a logistics controller to be deterministic and complete:

Definition 4 *(Deterministic I/O Automata). A deterministic I/O automaton is an I/O Automaton (Q, S, Σ, I, O, Ω, F) with*

- *exactly one initial state ($S = \{s\}$, $s \in Q$);*
- *no outgoing transitions from the final states: $\nexists_{(q_1,i,o,q_2)\in\Omega}\ q_1 \in F$;*
- *and branches that are only based on different outcomes of the same event: $\forall_{d_1,d_2\in\Omega,q_1,q_2,q_3\in Q,i_1,i_2\in I,o_1,o_2\in O}\ d_1 = (q_1, i_1, o_1, q_2),\ \ d_2 = (q_1, i_2, o_2, q_3),\ \ d_1 \neq d_2 \Rightarrow \exists_{e\in\mathbb{E},u_1,u_2\in\mathbb{O}}\ i_1 = (e, u_1), i_2 = (e, u_2) \wedge (e, u_1), (e, u_2) \in \gamma \wedge u_1 \neq u_2$.*

Note that the restriction on branches makes sure that we do not have multiple outgoing transitions of any state with the same label. The I/O automaton of the running example (Figure 3) adheres to this. Furthermore, determinism prohibits multiple outgoing transitions from any state with empty input labels (λ).

A complete I/O automaton is defined as follows.

Definition 5 *(Complete I/O Automata). An I/O automaton is called* complete *if at every branching state in the automaton in which an event is processed, there exists exactly one outgoing transition for each outcome of the processed event.*

The formalism introduced so far allows us to precisely define the supervisory-controller specification that our approach takes as input.

Definition 6 *(Supervisory Controller Specification). A supervisory controller specification is an activity specification (Definition 3) with a complete and deterministic I/O automaton specifying the logistics.*

Note that the specified supervisory controller of the running example does indeed have a complete and deterministic I/O automaton specifying the logistics.

3.2 | System Behavior

Any possible behavior of a system is captured by an accepted word of the I/O automaton in its specification. A behavior corresponds to a sequence of activities that the system executes in response to a sequence of event outcomes that are received from the plant. An activity specification with event feedback (Definition 3) and a deterministic I/O automaton (Definition 4) consist of deterministic parts, which are sequences of activities that emit events, and non-deterministic parts, which are the event outcomes that are received from the plant and determine the activities that follow. When an event outcome is received, the sequence of activities corresponding to the outcome is given by a path in the automaton that starts with a transition that captures that event outcome as input action. Therefore, the complete behavior the system exhibits in a particular execution depends on the event outcomes that are received from the plant.

To define system behavior, we first define a word in an I/O automaton as follows.

Definition 7 *(Words and Language of I/O Automata). A* word *in an I/O automaton Y is a finite sequence of pairs $(i, o) \in I \times O$. The word $w = (i_1, o_1), (i_2, o_2), \ldots, (i_j, o_j) \in (I \times O)^*$ is* accepted *by Y if there exists a sequence $q_0, q_1, \ldots, q_j \in Q^*$ of states such that $q_0 \in S$ and $(q_y, i_{y+1}, o_{y+1}, q_{y+1}) \in \Omega$, for all $0 \leq y < j$, and $q_j \in F$. The language $\mathcal{L}(Y)$ is the set of all words accepted by Y.*

An accepted word in the automaton gives a sequence of activities. For instance, the accepted word (λ, Act_1), (λ, Act_2), $((e, u_2), Act_4)$ of the I/O automaton of Figure 3 provides the sequence Act_1, Act_2, Act_4. The activity framework provides an operator, called the *sequencing operator* (Definition 8 given below), to combine a sequence of activities into a single activity, giving the same order and timing of action and event nodes. We use this operator to convert a behavior to a single activity (later defined in Definition 11). This facilitates the implementation of the execution engine.

Figure 4 shows the results of the sequencing operator sequencing Act_1, Act_2, and then Act_4 of our running example, after processing event e with outcome u_2. The final result, activity $Act_{1.2.4}$, corresponds to the accepted word $(\lambda, Act_1), (\lambda, Act_2), ((e, u_2), Act_4)$ of the I/O automaton (Figure 3). Intuitively, the operator removes the release nodes of the left-hand side activity and the claim nodes of the right-hand side activity, and adds dependencies from nodes that precede the removed release nodes to nodes that succeed the corresponding claim node based on the resource that they claim or release. These claim and release nodes that are removed are colored red in $Act_{1.2}$ and Act_4.

Recalling from Definition 1, we have two constraints to make sure that (VIII) claim nodes have no dependency on any other node and (IX) no nodes are dependent on release nodes. To illustrate the reason for this on our running example, looking at Figure 4, we observe that when sequencing two activities $Act_{1.2}$ and Act_4, dependencies between nodes from $Act_{1.2}$ that violate (IX) and dependencies between nodes from Act_4 that violate (VIII) do not exist in $Act_{1.2.4}$. The required dependencies are captured by new dependency relations between predecessors of the release nodes of the first activity and successors of the claim nodes of the second.

Note the dependencies from n_5 to n_7, and from n_5 to the release node of r_3 in $Act_{1.2.4}$, drawn in red color. These are added to capture the causality relation due to the outcome of e, produced by node n_5 and the corresponding

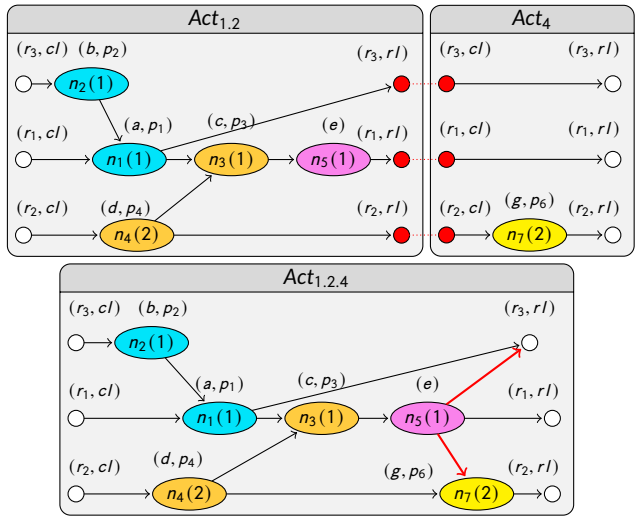


FIGURE 4 Activities $Act_{1.2}$, Act_4 and $Act_{1.2.4}$

sequence of activities that are executed in response (in this case, Act_4). In general, when an event is processed, a dependency is added from the node that emits that event to every succeeding node of all the claim nodes of the next activity to capture this causality. This means that all nodes of the sequence that occurs in response to the outcome have to wait until the outcome is resolved before they can execute.

The operator also enforces Constraint (XI) for every event, which requires that multiple instances of the same event must be sequentially ordered. For this purpose, it adds dependencies from every node that emits an event in the left-hand side activity to every node that emits that same event in the right-hand side activity. This is not visible in the example, because there only a single event is emitted. This is facilitated by strengthening Constraints (II) and (III) compared to [7]. In general, an activity may emit multiple instances of the same event. When we process an outcome of that event, for reasons of determinacy, we need to take into account which instance of the event is being processed.

Now that we have intuitively explained how the sequencing operator works, we define it formally. To allow multiple instances of the same activity to be sequenced without confusing the nodes in their DAGs, we use superscripts to create unique names for the nodes of the composed activities. If an event outcome is being processed, the operator also needs to capture the dependencies that arise between the event node of the left-hand activity that emits the event and the nodes of the right-hand activity that are executed after processing that event. We modify the sequencing operator of [7] to be able to capture these dependencies as follows.

Definition 8 *(Sequencing Operator) Given two activities* $A_1 = (Nod_1, \rightarrow_1, \mathcal{M}_1, \mathcal{T}_1)$ *and* $A_2 = (Nod_2, \rightarrow_2, \mathcal{M}_2, \mathcal{T}_2)$*, an event* e *being processed, and* k *denoting the instance number of* e *being processed. Let the sets of release nodes in* Nod_1*, and claim nodes in* Nod_2 *be* $rl_1 = \{n_1 \in Nod_1 \mid \exists_{r \in \mathbb{S}}\ \mathcal{M}_1(n_1) = (r, rl)\}$ *and* $cl_2 = \{n_2 \in Nod_2 \mid \exists_{r \in \mathbb{S}}\ \mathcal{M}_2(n_2) = (r, cl)\}$*, respectively. We define* $A_1;_{(e,k)} A_2$ *as the activity* $A_{1.2} = (Nod_{1.2}, \rightarrow_{1.2}, \mathcal{M}_{1.2}, \mathcal{T}_{1.2})$*, with:*

$$\begin{aligned} Nod_{1.2} = &\{n^1 \mid n \in Nod_1 \wedge n \notin rl_1\} \cup \\ &\{n^2 \mid n \in Nod_2 \wedge n \notin cl_2\} \end{aligned}$$

$$
\begin{aligned}
\rightarrow^{S}_{1.2} = \{&(n_i^1, n_j^1) \mid n_i \rightarrow_1 n_j \wedge n_j \notin rl_1\} \cup \\
&\{(n_i^2, n_j^2) \mid n_i \rightarrow_2 n_j \wedge n_i \notin cl_2\} \cup \\
&\{(n_1^1, n_2^2) \mid (\exists_{n_{rl} \in rl_1}\ n_1 \rightarrow_1 n_{rl}) \wedge \\
&\ (\exists_{n_{cl} \in cl_2}\ n_{cl} \rightarrow_2 n_2)\} \cup \\
&\{(n_1^1, n_2^2) \mid \exists_{e' \in \mathbb{E}}\, \mathcal{M}_1(n_1) = e' \wedge \mathcal{M}_2(n_2) = e'\} \\
\rightarrow^{P}_{1.2} = \{&(n^1_{e_k}, n_j^2) \mid n_{e_k} \in Nod_1 \wedge \mathcal{M}_1(n_{e_k}) = e \wedge \\
&\ n_j \in Nod_2 \wedge \exists_{n_{cl} \in cl_2}\ n_{cl} \rightarrow_2 n_j \wedge \\
&\ |\{n_e \mid \mathcal{M}_1(n_e) = e \wedge n_e \rightarrow_1^+ n_{e_k}\}| = k - 1\} \\
\rightarrow_{1.2} = {} & \rightarrow^{S}_{1.2} \cup \rightarrow^{P}_{1.2} \\
\mathcal{M}_{1.2} = \{&(n^1, \mathcal{M}_1(n)) \mid n \in Nod_1 \wedge n \notin rl_1\} \cup \\
&\{(n^2, \mathcal{M}_2(n)) \mid n \in Nod_2 \wedge n \notin cl_2\} \\
\mathcal{T}_{1.2} = \{&(n^1, \mathcal{T}_1(n)) \mid n \in Nod_1 \wedge n \notin rl_1\} \cup \\
&\{(n^2, \mathcal{T}_2(n)) \mid n \in Nod_2 \wedge n \notin cl_2\}
\end{aligned}
$$

If there are no events being processed when sequencing activities A_1 and A_2, then we denote the operation with $A_1.A_2$, yielding activity $A_{1.2} = (Nod_{1.2}, \rightarrow^{S}_{1.2}, \mathcal{M}_{1.2}, \mathcal{T}_{1.2})$. For sequencing the empty activity, we have for any activity A, $A.\epsilon = \epsilon.A = A$.

Note that in the final clause of the dependencies $\rightarrow^{S}_{1.2}$, we add dependencies between all pairs of event-emitting nodes from both activities, although adding a single dependency from the last node that emits the event in the left-hand side activity to the first one in the right-hand side activity would suffice, yielding an equivalent result, since the event emissions within both activities are already ordered. However, this would lead to a more complicated definition.

When processing an event, the sequencing operator uses the event that is being processed and its instance number (k) to capture the causality dependencies due to the event processing, as explained earlier.

We define a function, called the *behavior activity function*, that gives the activity of a behavior using the sequencing operator. We let τ be the empty sequence (we use it for empty sequences of different types).

Definition 9 *(Behavior Activity Function). Given an I/O automaton Y, the behavior activity function $\mathcal{B} : (I \times O)^* \rightarrow \mathbb{ACT}$ is recursively defined as follows, for $w \in (I \times O)^*$ and $(e, u) \in \mathbb{E} \times \mathbb{O}$,*

$$
\begin{cases}
\mathcal{B}(\tau) = \epsilon \\
\mathcal{B}(w(\lambda, o)) = \mathcal{B}(w).o \\
\mathcal{B}(w((e, u), o)) = \mathcal{B}(w);_{(e, |w|^e + 1)}\ o
\end{cases}
$$

where $|w|^e$ denotes the number of times event e is processed in word w.

Note that the sequencing operator captures the result of $|w|^e$, which is a number, in the variable k in the sequencing operation $A_1;_{(e,k)}\ A_2$ as defined in Definition 8 because the operator works on activity sequences, and not on I/O sequences.

We use the behavior activity function and the sequencing operator to construct the activity of a behavior (explained later in Definition 11). In the case of our running example, we obtain $\mathcal{B}((\lambda, Act_1), (\lambda, Act_2), ((e, u_2), Act_4)) = Act_{1.2.4}$ as shown in Figure 4, as follows. The function is recursively evaluated, using the third case in the above definition, for sequencing $Act_{1.2}$ with activity Act_4. In this operation, it processes instance $k = |(\lambda, Act_1), (\lambda, Act_2)|^e + 1 = 0 + 1 = 1$ of event e, as no instances of e are processed in Act_1 or Act_2 and the first instance is processed in Act_4. The recursive evaluation of $\mathcal{B}((\lambda, Act_1), (\lambda, Act_2))$ yields, following cases 2 and then 1 in the above definition, the sequence $Act_1.Act_2 = Act_{1.2}$ of activities to which Act_4 is sequenced. Hence, the result is $Act_{1.2};_{(e,1)} Act_4 = Act_{1.2.4}$. We describe in detail how this function is used for activity execution on a physical plant in Section 6.

An activity may emit multiple events with their outcomes resolved by the plant. However, the deterministic I/O automaton only processes a single event at each branching state. This imposes an order in which these event outcomes are processed which is given by the corresponding word in the automaton. The sequence of (event, outcome) pairs of a behavior corresponds to the order in which event outcomes are processed in the accepted word of the I/O automaton. We provide a function that extracts this sequence from any given accepted word of the automaton as defined below.

Definition 10 *(Processed Events Function). Given an I/O automaton Y, the processed events function* $\mathcal{V} : (I \times O)^* \rightarrow (\mathbb{E} \times \mathbb{O})^*$ *is recursively defined as follows, for* $w \in (I \times O)^*$ *and* $(e, u) \in \mathbb{E} \times \mathbb{O}$,

$$\begin{cases} \mathcal{V}(\tau) = \tau \\ \mathcal{V}(w(\lambda, o)) = \mathcal{V}(w) \\ \mathcal{V}(w((e, u), o)) = \mathcal{V}(w)(e, u) \end{cases}$$

To illustrate this function, we use our running example, where $\mathcal{V}((\lambda, Act_1),(\lambda, Act_2),((e, u_2), Act_4))) = (e, u_2)$. $\mathcal{V}((\lambda, Act_1),(\lambda, Act_2),((e, u_2), Act_4)))$ is evaluated, following the third case of the recursive definition, as $\mathcal{V}((\lambda, Act_1), (\lambda, Act_2))$ followed by (e, u_2). Applying the second case twice gives $\mathcal{V}((\lambda, Act_1), (\lambda, Act_2)) = \mathcal{V}((\lambda, Act_1))) = \mathcal{V}(\tau)$, which by the first case equals the empty sequence τ. The overall result is therefore (e, u_2), representing the emission of event e with outcome u_2.

With the sequencing operator and the language of the automaton defined, we now define the behavior of a system modeled in the activity framework with event feedback as follows.

Definition 11 *(Specified System Behavior). Given an activity specification (Acts,* $\mathbb{E}$, $\mathbb{O}$, γ, *Y), the specified system behavior P given a sequence of event outcomes provided by the plant is a word in* $\mathcal{L}(Y)$. *Such a word gives a sequence* $Act_1, Act_2, \ldots, Act_j$ *of activities, and a corresponding sequence* $U^P = (e_1, u_1), (e_2, u_2), \ldots, (e_i, u_i) \in (\mathbb{E} \times \mathbb{O})^*$ *of (event, outcome) pairs that lead to that sequence of activities.* $Act^P = \mathcal{B}(P)$ *is the activity of behavior P (Definition 9), and* $U^P = \mathcal{V}(P)$ *is the sequence of (event, outcome) pairs of P (Definition 10).*

3.3 | Model System Timing

With system behavior defined, now we focus on system timing. Whenever we execute an activity, it takes time for its resources to execute the individual actions. This means that executing an activity affects the timing at which system resources are available. In the activity framework, the state of the system constitutes the availability timing of each of the resources. While the I/O automaton captures the possible sequences of activities, it does not capture the timing of availability of system resources at each state. This is determined by the activities. We capture the resource state by a function $\bar{x}$ that maps each resource to the availability time of that resource. In the initial state $\bar{x}(0)$, all resources are assumed to be available at time 0. This state is denoted as $\bar{0}$.

When we execute an activity, the resource availability function is updated to represent the state of the system after executing the activity, claiming the resources as soon as they are available in the starting state and releasing them when all actions on them have been performed. We define model system timing as the timings of action and event nodes as follows. These definitions conform to the max-plus semantics of an activity model with event feedback as defined in [12].

Definition 12 *(Model System Timing). Given an activity* $(Nod, \rightarrow, \mathcal{M}, \mathcal{T})$ *and resource availability function* $\bar{x}$*, the specified start time* $S(n, \bar{x})$ *and completion time* $C(n, \bar{x})$ *of node* $n \in Nod$ *are defined by*

$$S(n, \bar{x}) = \begin{cases} \bar{x}(r) & \text{if } n = (r, cl) \\ max_{n' \in Pred(n)} C(n', \bar{x}) & \text{otherwise} \end{cases}$$

$$C(n, \bar{x}) = S(n, \bar{x}) + \mathcal{T}(n)$$

Wherever $\bar{x}$ *is known from context, we abbreviate* $S(n, \bar{x}), C(n, \bar{x})$ *to* $S(n), C(n)$*. For* $r \in \mathbb{R}$*, we define the functions* $S + r$ *and* $C + r$ *as follows*

$$(S + r)(n) = S(n) + r$$

$$(C + r)(n) = C(n) + r$$

Considering activity Act_1 of Figure 2 as an example, executing this activity from initial state $\bar{0}$ results in resource release times (completion times of the resource release nodes) for r_1, r_2, and r_3 of 2, 0, and 2, respectively.

Model system timing for a behavior P (Definition 11), denoted by $S^P(\bar{x})$ and $C^P(\bar{x})$, can be computed using the activity of a behavior and a given resource availability function. Wherever $\bar{x}$ is known from context, we abbreviate to S^P and C^P, respectively.

4 | EXECUTION RELATION TO PRESERVE

In this section, we define the relation that we aim to establish between the behavior and timing prescribed by the specified model and the behavior and timing exhibited by the physical execution. We first define the timing relation. Intuitively, we want our execution to adhere to the timing that the model specifies. This means the start times and the completion times of actions and events in the model must be the same as their counterparts in execution, allowing, however, for some error bound, which we define in this section. We first define notations for start and completion times of actions and events as well as their durations in a physical execution.

Definition 13 *(Physical Execution Timings). Given an activity* $(Nod, \rightarrow, \mathcal{M}, \mathcal{T})$*, for action nodes* $\mathcal{M}(n) \in \mathbb{A} \times \mathbb{P}$*, the* execution start time*,* $S'(n)$*, is the wall clock time when the action starts on the real peripheral of the plant, the* execution completion time*,* $C'(n)$*, is the wall clock time when the action completes on the real peripheral of the plant, and the* execution duration *is* $\mathcal{T}'(n) = C'(n) - S'(n)$*. For event nodes* $\mathcal{M}(n) \in \mathbb{E}$*, the* event trigger time*,* $S'(n)$*, is the wall clock time at which the sampling of data starts, the* event emission time*,* $C'(n)$*, is the wall clock time at which the sampling of data completes, and the* event delay *is* $\mathcal{T}'(n) = C'(n) - S'(n)$*.*

The activity framework defines start and completion times for actions, but there is no clearly defined relation between model time and physical time. Because we want to ensure that the timing of the physical execution ad-

heres to the timing of the model, we need to establish a relation between timing of actions and events in the model ($S(n), C(n), \mathcal{T}(n)$, see Definition 12) and their timing in physical execution ($S'(n), C'(n), \mathcal{T}'(n)$). We do so by first defining the start time of the physical execution as follows.

Definition 14 *(Execution Start Time). The start time of the physical execution, denoted by Ψ, is the physical wall clock time, given by the user, at which the execution should start.*

Executing actions in reality has complications that the activity framework abstracts from. This includes initializing the execution parameters and configuring the controller that is used to execute the model. It also includes delays introduced by the laws of physics since it takes some time for action execution commands to travel inside wires and reach the peripheral. Finally, the controller has to take some time to compute the action and event nodes to perform.

To address the physical delays in wires and the controller computation time, we define the following and show how we deal with them in Section 6.

Definition 15 *(Execution Delay). Given an activity $(Nod, \rightarrow, \mathcal{M}, \mathcal{T})$ and assuming $\bar{x}(0) = \bar{0}$, for all $n \in Nod$, such that $\mathcal{M}(n) \in \mathbb{A} \times \mathbb{P} \cup \mathbb{E}$, the* execution delay *of n is*

$$\delta(n) = S'(n) - S(n) - \Psi$$

In Section 6, we propose a plant model and an execution engine, that together form an execution architecture. Assuming the inevitable innate delays are bounded, this architecture provides an upper bound $\mathbb{D}_{\mathbb{A}}$ on these execution delays such that

$$\forall_{n \in Nod, \mathcal{M}(n) \in \mathbb{A} \times \mathbb{P} \cup \mathbb{E}}\ \delta(n) \leq \mathbb{D}_{\mathbb{A}} \tag{1}$$

and uses the model time as a lower bound such that

$$\forall_{n \in Nod, \mathcal{M}(n) \in \mathbb{A} \times \mathbb{P} \cup \mathbb{E}}\ S'(n) \geq S(n) + \Psi$$

With these definitions, we define the relation that we want to establish between the timings of the action and event nodes in the model and their corresponding timings in the physical execution.

Definition 16 *(Timing Relation and Timing Preservation). Given two pairs of start and completion functions S_1, C_1 and S_2, C_2 all defined over the same domain, the set Nod of nodes, we denote their execution timing relation as*

$$S_1, C_1 \sim S_2, C_2 \iff \forall_{n \in Nod} S_1(n) \leq S_2(n) \leq S_1(n) + \mathbb{D}_{\mathbb{A}} \wedge C_2(n) \leq C_1(n)$$

That is, start times S_2 are not earlier than start times S_1 and not later than S_1 plus the earlier defined execution delay bound; completion times C_2 are not later than completion times C_1. If $S_1, C_1 \sim S_2, C_2$, then S_2, C_2 is said to preserve the timing *of S_1, C_1.*

The defined timing preservation relation can be applied to any two sets of start and completion times of a set of nodes. The intended use though is to apply timing preservation to model times and execution times of a physical execution, where the model times take the role of S_1 and the physical execution times take the role of S_2.

Now we define the relation that we want to preserve between the system behavior (Definition 11) and the corresponding behavior that the execution engine executes.

Recall that a behavior is an accepted word of the I/O automaton and the activity and the sequence of (event, outcome) pairs that are derived from it. A deterministic I/O automaton (Definition 4) is deterministic up to states where alternative event outcomes give the next deterministic path. The behavior that is executed is one that corresponds to an accepted word of the I/O automaton and is given by the sequence of event outcomes resolved at runtime. Note that in a deterministic I/O automaton, every accepted word corresponds to a unique path in the automaton.

To define behavior preservation, we first have to note that the specified behavior of a system (Definition 11), when executed, boils down to action and event nodes that are executed and can be observed in the plant, the outcomes of the events that were given by the plant, and the order in which the outcomes were processed. Resource nodes are not observable in the plant and are used as a model abstraction that enables expressing constraints on the model to ensure correctness of behavior and validity of execution. Therefore, executed system behavior, and consequently, behavior preservation do not take resource nodes into account.

Definition 17 *(Executed System Behavior). The executed system behavior is a set Nod' of all action and event nodes that are executed and their start and completion times S', C' (Definition 13), and a sequence U' of processed (event, outcome) pairs that is received from the plant.*

Intuitively, preserving a behavior is executing exactly all the nodes of its activity while preserving all the dependencies between all the action and event nodes and respecting the event outcomes that are resolved at runtime and processing them in the order that the behavior specifies. Since the event outcomes are determined at execution time by the plant, the behavior that the execution engine preserves is also determined at runtime. Because the I/O automaton is complete (Definition 5), all possible outcomes of every event that is emitted and resolved during execution lead to an accepted word, and therefore, to a specified system behavior. We formally define behavior preservation as follows.

Definition 18 *(Behavior Preservation). Given an activity specification (Acts, $\mathbb{E}$, $\mathbb{O}$, γ, Y), and starting state $\bar{x}$, let P be a specified behavior (Definition 11) with (Nod_P, $\rightarrow_P$, $\mathcal{M}_P$, $\mathcal{T}_P$) its corresponding activity, and U^P its sequence of (event, outcome) pairs. We say P is preserved if and only if all action and event nodes $n \in Nod_P$ are executed, no other actions or events are executed, the same sequence of event outcomes are received from the plant and processed, and for every dependency $n_1 \rightarrow_P n_2$, we have $C'(n_1) \leq S'(n_2)$.*

The primary goal of the execution engine and the execution architecture that we present in Section 6 is to preserve any behavior (Definition 18) and timing (Definition 16) of an activity specification by respecting the event outcomes that correspond to the behavior, executing exactly all event and action nodes of the activities that it is composed of, and preserving node dependencies by preserving the start and completion times of those nodes.

5 | PROBLEM STATEMENT

With the precise definitions of a supervisory controller specification (Definition 6) and its operational characterization, as well as the formalization of the physical timing and behavior of a plant, we can precisely phrase the problem addressed by the proposed execution architecture, which is how to execute the supervisory controller of a plant, specified as an activity specification with event feedback (Definition 3) such that any behavior that is executed respects the order and outcomes of events received from the plant, preserving timing (Definition 16) and behavior (Definition 18).

In the next section, we first present the assumptions on the plant and on the execution engine that, together, form the assumptions that we need for the execution architecture to preserve timing and behavior. In Section 6.5, we prove that it does preserve the timing relation (Definition 16) for any behavior. Preservation of node dependencies (Definition 18) then follows from that. We also explain how it respects event outcomes and preserves the set of action and event nodes that it executes.

6 | EXECUTION ARCHITECTURE

In this section, we define our execution architecture composed of the supervisory controller and the plant (Figure 1). We first describe what assumptions we impose on the plant, then we describe the execution engine. Afterward, we show our approach using the running example. Finally, we discuss the boundedness of the delays associated with executing a model, and we prove how our architecture preserves the relation between the specification and execution.

6.1 | Plant

For our execution engine to work properly, we need two assumptions on the plant.

Assumption 1 *(Execution Duration Bounds) The plant performs the set of actions with known bounds on their execution durations.*

Assumption 2 *(Execution Delay Bounds) There is a known bound on the amount of time that the plant takes to initiate actions or to process event data, and observe node completions. We denote the* Execution Delay Bound *with* $\mathbb{D}_{\mathbb{A}}$.

These assumptions are reasonable for any CPS unless there is a failure in the system. Any failure that violates any of the above assumptions can be detected by the execution engine. It then provides an error message.

6.2 | Execution Engine Description and Formalization

The execution engine is configured with an activity specification (Definition 3). When configured, it constitutes the supervisory controller. It has three layers each corresponding to the different layers in the model. At the top of the execution engine, we have the Logistics Controller (LC). This layer is configured with the logistics controller specification (the I/O automaton) and is responsible for executing it, ensuring that the right activities are executed, in the right order, in accordance with the received event outcomes, as specified. In the middle layer of the engine, we have the Activity Controller (AC), which is configured with activity specifications and is responsible for the execution of activities, respecting specified dependencies between actions and on the resources. At the bottom layer of the engine, we have the Action Controller (aC), which is configured with action and event node specifications and executes them on the plant using a time-triggered approach.

Recall that the model consists of deterministic parts, which are sequences of activities that emit events, and non-deterministic parts, which are the event outcomes that are received at runtime from the plant and determine the activities that follow. The execution engine iteratively executes deterministic parts and processes the non-deterministic event outcomes from the plant.

The deterministic I/O automaton (Definition 4) captures the deterministic parts of the model with paths through the automaton with all intermediate states, if any, only having one outgoing transition. These paths have no repeated

states except possibly the first and last states, and they end with a state that is either a final state or a state with multiple outgoing transitions. The states with multiple outgoing transitions capture the non-deterministic part of the behavior with branches that correspond to the different outcomes of the same event. So the overall behavior is deterministic up to event outcomes. The branching states are where the controller decides on the transition to take based on the event outcome that has been received. We call these states *decision states*.

Definition 19 *(Decision State). Given a deterministic I/O automaton $(Q, S, \Sigma, I, O, \Omega, F)$, a* decision state *is a state in Q with more than one outgoing transition.*

Going back to the I/O automaton of the running example (Figure 3), state q_2 is a decision state (and it is the only one). The sequences that we execute correspond to a *path* in the logistics automaton.

Definition 20 *(Path). Given an I/O automaton $(Q, S, \Sigma, I, O, \Omega, F)$, with $q_k \in Q$, $i_l \in I$, and $o_l \in O$, for $0 \leq k \leq n$, $1 \leq l \leq n$, and some $n \in \mathbb{N}$, a* path *is a sequence of 4-tuples $\rho = (q_0, i_1, o_1, q_1), (q_1, i_2, o_2, q_2), \ldots, (q_{n-1}, i_n, o_n, q_n)$ such that $(q_{j-1}, i_j, o_j, q_j) \in \Omega$ for each $1 \leq j \leq n$.*

For instance, $(q_0, \lambda, Act_1, q_1), (q_1, \lambda, Act_2, q_2), (q_2, (e, u_1), Act_3, q_0)$ is a path in the I/O automaton of our running example.

The execution engine divides any path in the automaton that it is executing into its constituent deterministic parts called *decision paths*.

Definition 21 *(Decision Path). Given a deterministic I/O automaton $(Q, S, \Sigma, I, O, \Omega, F)$, a path $(q_0, i_1, o_1, q_1), (q_1, i_2, o_2, q_2), \ldots, (q_{n-1}, i_n, o_n, q_n)$ is a* decision path *if q_0 is either an initial state or a decision state, and q_n is either a final state $(q \in F)$ or a decision state, and $q_1, \ldots, q_{n-1}$ are not decision states.*

The decision paths of the running example are $\rho_1 = (q_0, \lambda, Act_1, q_1), (q_1, \lambda, Act_2, q_2)$, $\rho_2 = (q_2, (e, u_1), Act_3, q_0), (q_0, \lambda, Act_1, q_1), (q_1, \lambda, Act_2, q_2)$, and $\rho_3 = (q_2, (e, u_2), Act_4, q_3)$.

Before the execution of a model starts, we have an initialization phase where the execution engine is configured with the model. It starts when the LC reads the transition labels of the I/O automaton starting from the initial state until it reaches a decision state or a final state and then forms a decision path ρ. The LC then sends the sequence of activity labels corresponding to ρ to the AC. We assume that logistics automata do not have states without paths to a final state. Such states are redundant in specifying behavior, and they complicate model-driven execution.

Assumption 3 *(No Blocking States). All non-final states in the logistics automaton have a path to a final state.*

When the activity controller AC receives the sequence of activity labels from the LC, it sequences them (based on Definition 8). This essentially means that the AC executes a single activity. This facilitates implementation. The AC computes the start and completion times of every node in the executed activity based on the availability of resources and the completion time of the event node that they depend on (given by the behavior activity function, Definition 9). Then, the AC sends the labels of the action nodes and the event nodes of the resulting activity and their specified start and completion times to the aC to be executed in a time-triggered manner. The initialization phase ends when the aC has received and prepared its first set of nodes and is ready to execute them. The delay caused by this initialization is not different from the initialization delay of other controllers used in CPS. Since we want to be precise, making all requirements explicit, we assume the following.

Assumption 4 *(Execution Start Time). Ψ, is a moment in time far enough in the future to allow the initialization to take place before it.*

Another assumption in our approach is that the specified model timing of every action is at least equal to the physical action execution duration *plus* the execution delay upper bound $\mathbb{D}_{\mathbb{A}}$. That is, model timings are conservative with respect to physical action execution times. This ensures that the time-triggered execution of the aC keeps the timing and order of actions prescribed by the specification. Formally, we write this assumption as follows.

Assumption 5 *(Conservative Action Durations) For all action nodes n in an activity specification, $\mathcal{T}(n) \geq \mathcal{T}'(n) + \mathbb{D}_{\mathbb{A}}$.*

After the aC receives the node labels and their specified start and completion times from the AC, it sorts them based on their intended start time $S(n) + \Psi$. Then, it compares its clock ψ with the intended start time of each node it has received from the AC, and executes each action or event node when its time arrives ($\psi = S(n) + \Psi$). To execute event nodes, the aC translates data from the plant to event outcomes and sends those to the AC. The aC also observes the execution of action and event nodes on the plant to check whether they complete within their model-specified time (Assumption 5). If an action or event node should not complete before its specified completion time plus Ψ (this could only happen in violation of the assumptions), appropriate action can be taken, such as warning the user or logging the violation.

When the AC receives an event outcome from the aC, it notifies the LC of the outcome of the event as soon as possible. Then, the LC traverses the I/O automaton to the next decision state and sends the corresponding sequence of activity labels to the AC, as well as the event being processed and its instance (e and k of the sequencing operator, Definition 8). These steps repeat until we reach a final state at which point the execution terminates.

Considering the delays that impact the execution times, we let $\mathbb{D}_{\mathbb{E}}$ (event processing time) denote the maximum time that it takes for the execution engine to receive an event outcome, find the next decision path, sequence the corresponding activities, and prepare to execute the action and event nodes of that activity. Hence, $\mathbb{D}_{\mathbb{E}}$ denotes the maximum time required to process events.

To provide enough time for event processing, we assume that the specified delay of an event ($\mathcal{T}(e)$) is larger than the time that it takes to resolve ($\mathcal{T}'(e)$, Definition 13) and process ($\mathbb{D}_{\mathbb{E}}$) the outcome. If a specification does not satisfy this assumption, the controller will not have enough time to process the outcome, leading to the execution being delayed. Recalling $\mathbb{D}_{\mathbb{A}}$ from Equation 1, we formalize this assumption as follows.

Assumption 6 *(Conservative Event Delays) For all event nodes n in an activity specification, $\mathcal{T}(n) \geq \mathcal{T}'(n) + \mathbb{D}_{\mathbb{E}} + \mathbb{D}_{\mathbb{A}}$.*

Note that Assumptions 1-6 do not limit the use of our approach beyond what can be considered a typical system. Assumption 1 (Execution Duration Bounds) means that system actions eventually end and their duration is known. This covers the majority of CPS. The same can be said about Assumption 2 (Execution Delay Bounds) which informally means that the communication between the controller and the actuators is reliable with a known bound on its delay. The other assumptions are on the model specification. Assumption 3 (No Blocking States) ensures that the model has no redundant states. Such states, if present, can easily be removed from a model before execution. The other three assumptions ensure that a specified model meets the timing requirements of the plant. Assumption 4 (Execution Start Time) ensures that the user allows proper time for the controller to initialize before it can start any action. The last two assumptions ensure that model timings of action execution and event processing are conservative with respect to physical reality. We are explicit about our assumptions for completeness and to be precise.

6.3 | Example Execution

We now show how the execution of a supervisory controller specification works using the running example. During the initialization phase, the logistics controller LC is configured with I/O automaton Y (Figure 3) and the activity controller AC is configured with the activities (Figure 2). The LC starts from the initial state q_0 of I/O automaton Y and traverses the automaton until it reaches a decision state. Since q_1 has only one outgoing transition and is not a final state, LC continues to q_2 where there are multiple outgoing transitions with event outcomes. Therefore, q_2 is a decision state where the LC stops traversing the automaton and sends the activity-label sequence Act_1, Act_2 corresponding to decision path ρ_1 introduced earlier to the AC.

At this point, the AC obtains $Act_{1,2}$ using the sequencing operator and sends the labels of the action and event nodes and their start and completion times to the action controller aC. The aC now has the set $\{(n_1, \Psi + 1, \Psi + 2), (n_2, \Psi, \Psi + 1), (n_3, \Psi + 2, \Psi + 3), (n_4, \Psi, \Psi + 2), (n_5, \Psi + 3, \Psi + 4)\}$ of action and event nodes to execute, where each triple in the set is an action or event node with its intended start and completion times $\Psi + S(n)$ and $\Psi + C(n)$. The aC sorts these nodes based on their intended start times. The first nodes to be executed are nodes n_2 and n_4. When the clock of the aC ψ reaches Ψ, it executes n_2 and n_4. When $\psi = \Psi + 1$, it executes n_1. Note that Act_1 claims both r_1 and r_3 while n_1 executes, but n_1 only executes on peripheral p_1 of r_1 and r_3 remains claimed until the completion time of n_1. When $\psi = \Psi + 2$, it executes n_3. When $\psi = \Psi + 3$, it samples the sensor corresponding to the event e, translates the data to an outcome of e, and sends it back to the AC. The operation of the execution engine explained here is illustrated in Figure 5.

When the action controller aC notifies the activity controller AC of the outcome of event e, the AC notifies the LC of the received outcome and the LC traverses the corresponding path in the automaton to obtain the next activity-label sequence to send to the AC. This next sequence is determined by the outgoing transitions of q_2 that process event e emitted by Act_2 with two possible outcomes, u_1 and u_2. If we assume that the outcome u_1 is received, the LC sends the activity-label sequence corresponding to decision path ρ_2. This brings the execution to a repeating sequence of activities (Act_3, Act_1, Act_2) being executed that lasts as long as the outcome of e is u_1 every time it is emitted. If u_2 is received, the LC sends the activity sequence corresponding to decision path ρ_3. In this case, the activity Act_4 is executed after which the execution halts. Note that the AC is made aware by the aC what node emitted the event e. Therefore, it has the necessary information to correctly sequence the following activities with their dependencies to the event node (depicted in red in Figure 4).

Figure 5 shows the execution of the decision path ρ_1, receiving event outcome u_1, and consequently executing ρ_2 (for brevity, ρ_2 is only partially depicted). There are three rows corresponding to system resources r_1, r_2, r_3. Solid blocks on the upper side of each row depict model durations of action and event nodes, while hatched ones on the lower side represent execution durations. Solid blocks on the lower side at the start and end of each hatched block represent execution delays which are together bounded by $\mathbb{D}_{\mathbb{A}}$. Since Act_1 claims both r_1 and r_3 while n_1 executes, it is depicted by two sets of blocks each time that n_1 executes, one on r_1 where it executes (a, p_1) (with a solid and a hatched block), and another on r_3 which is only claimed by Act_1 and no action executes on r_3 (hence, no hatched block). Based on Assumption 6, the aC sends the outcome of event e at time $C'(e)$ which occurs before $\Psi + 4 - \mathbb{D}_{\mathbb{E}}$ at the latest. Therefore, the execution engine has enough time to send the nodes of ρ_2 to the aC to be executed. Thus, we hide $\mathbb{D}_{\mathbb{E}}$ and prevent it from causing any delay in the execution of nodes, compared to the model. Nodes n_4 of Act_2 and n_6 of Act_3 execute after the completion time of event node e because they cannot start before e is resolved and therefore, there is an implicit dependency between them. The execution of n_6 of Act_3 means that n_2 of (the second instance of) Act_1 executes at $\Psi + 6$ (and consequently n_1 of Act_1 at $\Psi + 7$) because they depend on r_3, whereas n_4 executes at $\Psi + 4$ since all its dependencies are met at that time.

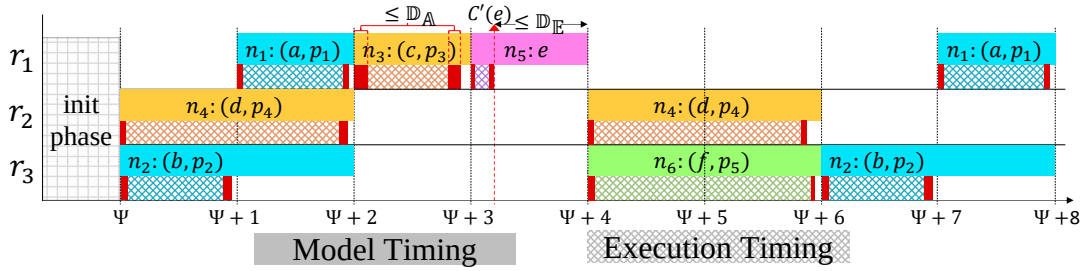


FIGURE 5 Partial execution Gantt chart of decision paths ρ_1 and ρ_2

6.4 | Delay Bounds

Our architecture has delays that are bounded by $\mathbb{D}_{\mathbb{E}}$ and $\mathbb{D}_{\mathbb{A}}$. Since the existence of bound $\mathbb{D}_{\mathbb{A}}$ is an assumed property of the plant, we determine the value of bound $\mathbb{D}_{\mathbb{E}}$. We start by showing that our restrictions on the I/O automaton and activities ensure that the number of activities and the number of nodes to be processed and executed in each iteration of the execution engine have a known bound. Then, we show that the timing relation (Definition 16) holds (Proposition 1).

The I/O automaton is deterministic with finite sets of states, input labels and output labels. The number of decision paths that the LC reads is finite. If the automaton has cycles, then since a cycle cannot contain any final state, and based on Assumption 3, it must have a branch that leads to a final state. Therefore, the state with that branch is a decision state where the LC stops reading the activity labels of that cycle and sends the decision path that it has obtained so far to the AC. Therefore, cycles do not cause the specification to have an infinite number of decision paths. This also makes the number of activity labels given by a decision path and sent to the AC finite and bounded by the longest decision path which can be determined from the specified I/O automaton. Because an activity has a finite number of nodes, the number of nodes to be executed at any time during the execution is finite. This number is bounded by the largest number of nodes in any decision path. Since when processing an event, the execution engine is dealing with a bounded number of these bounded ingredients of the specification, then the time to process an event must have a bound, which we call $\mathbb{D}_{\mathbb{E}}$. $\mathbb{D}_{\mathbb{E}}$ may be computed using the following equation:

$$\mathbb{D}_{\mathbb{E}} = \mathbb{D}_{Event} + \mathbb{D}_{LC} + \mathbb{D}_{AC} + \mathbb{D}_{aC}$$

where $\mathbb{D}_{Event}$ is the maximum time that it takes for an event outcome to be sent from aC to the AC and finally to LC, $\mathbb{D}_{LC}$ is the maximum time that the LC takes to read the largest decision path in the automaton and find the activity label sequence and send it to the AC, $\mathbb{D}_{AC}$ is the maximum time that the AC takes to sequence the largest activity sequence and find node start and completion times and send them to the aC, and $\mathbb{D}_{aC}$ is the maximum time that the aC takes to prepare the largest number of nodes in any decision path that it has received. $\mathbb{D}_{LC}$, $\mathbb{D}_{AC}$, and $\mathbb{D}_{aC}$ depend on the size of the specification, while clearly, all these values, including $\mathbb{D}_{Event}$, depend on the speed of the processing platform and the implementation.

The LC performs a number of operations to read transition labels of a decision path and send them to the AC that grows linearly with the number of transitions. Let $L(\rho)$ be the length (the number of transitions) of a decision path ρ in the I/O automaton. The computational complexity of the operations that the LC performs is $O(L(\rho))$, which means that the execution time of the operations grows linearly with the length of ρ. The AC performs the most complex

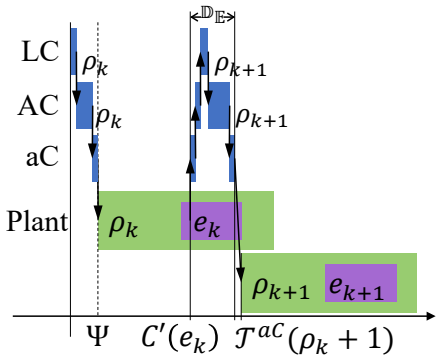


FIGURE 6 Execution chart of a sequence of decision paths

operations of the entire engine. Based on the activity labels received from the LC, it finds the DAG of each activity, which depends on $L(\rho)$. It then sequences all activities of ρ. The sequencing scales with the number of resources of the activities we sequence, $\mathcal{R}(Act)$, $Act \in Acts$, and the number of nodes in them, see $\rightarrow_{1.2}$ in Definition 8. The AC then sends node information to the aC, which is of complexity $O(L(\rho))$. Therefore, $\mathbb{D}_{AC} = O(L(\rho) \times |Acts| + L(\rho) \times \mathcal{R}(Act)^2 \times |Nod_{Act}|^2)$. Finally, for aC, we have $\mathbb{D}_{aC} = O(N(\rho)\log(N(\rho)))$ where $N(\rho)$ is the number of nodes in ρ, since it sorts the actions that it receives.

$\mathbb{D}_{\mathbb{E}}$ can be computed if the processing platform gives hard real-time guarantees on its timings. For instance, by measuring how long the LC takes to perform its operations for every transition, including reading and sending, we can compute the time that it needs for every decision path. Using the longest decision path in the automaton, we can compute the bound $\mathbb{D}_{LC}$. If we use a processing platform without timing guarantees, $\mathbb{D}_{\mathbb{E}}$ may be determined through measurements.

6.5 | Time- and Behavior-Preservation Proof

Recall that the aC implements a time-triggered execution. For time-triggered executions in general, there are two cases to be made for execution start time of every node. Either it has received the node and is prepared to execute it when its time arrives, or it receives the node after its time arrives. We denote the time at which the aC has all nodes of a decision path ρ and is ready to execute them by $\mathcal{T}^{aC}(\rho)$ and formulate the above argument as follows.

$$\forall_{n \in Nod_\rho}\ S'(n) = max(S(n) + \Psi + \delta(n), \mathcal{T}^{aC}(\rho)) \tag{2}$$

To formally prove that our execution engine satisfies the timing relation (Definition 16) between model and execution, we first need to introduce a few notations in relation to the timing aspects of decision paths. Since decision paths give sequences of activities and activities have timing properties, we can derive the timing properties of a decision path based on the timing of its activity sequence, given a starting state. For that, we first introduce the start time of an activity.

Definition 22 *(Specified Start Time of an Activity). Given activity* $A = (Nod_A, \rightarrow_A, \mathcal{M}_A, \mathcal{T}_A)$ *and resource availability*

function $\bar{x}$, the specified start time of activity A is defined as

$$S(A, \bar{x}) = \begin{cases} min_r\ \bar{x}(r), & \text{if } Nod_A = \varnothing \\ min_{n \in Nod_A}\ S(n, \bar{x}), & \text{if } Nod_A \neq \varnothing \end{cases}$$

Wherever $\bar{x}$ is known from context, we abbreviate $S(A, \bar{x})$ to $S(A)$.

Based on the start time of the activity sequence of a decision path, we define the start time of a decision path as follows.

Definition 23 *(Specified Start Time of a Decision Path). Given decision path $\rho = (q_0, i_1, Act_1, q_1), (q_1, i_2, Act_2, q_2), \ldots, (q_{n-1}, i_n, Act_n, q_n)$ and resource availability function $\bar{x}$, the specified start time of decision path ρ is defined as*

$$S(\rho, \bar{x}) = S(Act_{1.2.\ \ldots\ .n}, \bar{x})$$

(with $Act_{1.2.\ \ldots\ .n}$ defined in Definition 8). Wherever $\bar{x}$ is known from context, we abbreviate $S(\rho, \bar{x})$ to $S(\rho)$. We refer to start and completion times of action nodes in ρ with S^ρ and C^ρ, respectively.

Given the notations above, now we prove that the execution engine satisfies the timing relation (Definition 16). We start by introducing a lemma that shows given our set of assumptions, the execution engine satisfies the execution relation for any path in the I/O automaton. Then, since any specified behavior is a path made of a sequence of decision paths in the automaton, we prove that the execution relation is kept by our execution engine for any activity specification with event feedback that satisfies our assumptions. We use Figure 6 in our proof, which depicts the Gantt chart of a specification.

Lemma 1 *For any path $\rho = \rho_1, \rho_2, \ldots, \rho_i$ in an I/O automaton where ρ_j are decision paths, for all $k \in \mathbb{N}, 1 \leq k \leq i$, we have $S^{\rho_k} + \Psi, C^{\rho_k} + \Psi \sim S'^{\rho_k}, C'^{\rho_k}$.*

Proof We break down the relation into its constituent inequalities according to Definition 16 and prove them individually. For all $n \in Nod_{\rho_k}$ and assuming starting state $\bar{x}_0$, we prove

$$S(n, \bar{x}_0) + \Psi \leq S'(n, \bar{x}_0) \tag{3}$$

$$S'(n, \bar{x}_0) \leq S(n, \bar{x}_0) + \Psi + \mathbb{D}_{\mathbb{A}} \tag{4}$$

$$C'(n, \bar{x}_0) \leq C(n, \bar{x}_0) + \Psi \tag{5}$$

The aC uses a time-triggered execution mechanism that, by design, does not execute any action too early. Therefore, Equation 3 follows trivially from Equation 2.

We prove Equations 4 and 5 by induction on the number of decision paths in ρ. For the base case $k = 1$, the execution starts after all three layers have performed their tasks on ρ_1 and by definition, we have $\Psi = S'(\rho_1)$. The aC has all nodes $n \in Nod_{\rho_1}$ that it needs to execute at time Ψ. This means that $\mathcal{T}^{aC}(\rho_1) \leq S(n) + \Psi$ for all $n \in Nod_{\rho_1}$. Based on the principles of time-triggered execution (Equation 2) and the upper bound on execution delays (Assumption 2), we get (for brevity, omitting $\bar{x}_0$ henceforth)

$$S'(n) \leq S(n) + \Psi + \mathbb{D}_{\mathbb{A}}$$

Therefore, Equation 4 holds for $k = 1$. Because node durations are conservative (Assumption 5) and they start execution within a given bound from their model time (Equation 4), we add these two inequalities together to prove Equation 5.

$$S'(n) + \mathcal{T}'(n) + \cancel{\mathbb{D}_{\mathbb{A}}} \leq S(n) + \Psi + \cancel{\mathbb{D}_{\mathbb{A}}} + \mathcal{T}(n)$$

By replacing the time durations $\mathcal{T}(n)$ and $\mathcal{T}'(n)$ by their start and completion times (Definition 12,13), we get

$$\cancel{S'(n)} + C'(n) - \cancel{S'(n)} \leq \cancel{S(n)} + \Psi + C(n) - \cancel{S(n)}$$

This means that Equation 5 also holds for $k = 1$.

Our induction hypothesis is that Equation 4 and Equation 5 hold for path k, and we prove the relation for the induction step to path $k + 1$. Let e_k be the event emitted by ρ_k, the outcome of which determines the next decision path. From the induction hypothesis, we know that $S'(e_k) \leq S(e_k) + \Psi + \mathbb{D}_{\mathbb{A}}$. Since event delays are conservative, by adding Assumption 6 to this equation, we get

$$S'(e_k) + \mathcal{T}'(e_k) + \cancel{\mathbb{D}_{\mathbb{A}}} + \mathbb{D}_{\mathbb{E}} \leq S(e_k) + \Psi + \cancel{\mathbb{D}_{\mathbb{A}}} + \mathcal{T}(e_k)$$

We replace the time durations $\mathcal{T}(n)$ and $\mathcal{T}'(n)$ by their start and completion times (Definition 12,13), and we get

$$\cancel{S'(e_k)} + C'(e_k) - \cancel{S'(e_k)} + \mathbb{D}_{\mathbb{E}} \leq \cancel{S(e_k)} + C(e_k) - \cancel{S(e_k)} + \Psi$$

Because $C(e_k) \leq S(\rho_{k+1})$, and $C'(e_k) + \mathbb{D}_{\mathbb{E}} = \mathcal{T}^{aC}(\rho_{k+1})$, we can write

$$\mathcal{T}^{aC}(\rho_{k+1}) \leq S(\rho_{k+1}) + \Psi$$

From Definition 23, we conclude that

$$\forall_{n \in Nod_{\rho_{k+1}}} S'(n) \leq S(n) + \Psi + \mathbb{D}_{\mathbb{A}}$$

This means that Equation 4 holds for $k + 1$. The proof for Equation 5 holding for $k + 1$ follows the same identities as for the base case. We conclude that Equations 4 and 5 hold for case $k + 1$. This concludes the induction proof, and consequently, the complete proof.

Proposition 1 *For any activity specification with event feedback (Definition 3), under Assumptions 1-6, let Ψ be the start time of the execution. If S^P, C^P are specified start and completion times, and S'^P, C'^P are the execution start and completion times of the nodes in a behavior P (Definition 11) of the specification, then $S^P + \Psi, C^P + \Psi \sim S'^P, C'^P$.*

We prove Proposition 1 using Lemma 1.

Proof To prove Proposition 1, we observe that any behavior P of an activity specification with event feedback is derived from an accepted sequence of decision paths in the automaton. Since Lemma 1 shows that any sequence of decision paths is executed on time, then P is executed on time. Therefore, $S^P + \Psi, C^P + \Psi \sim S'^P, C'^P$.

Note that preserving dependencies between nodes follows as a corollary of Proposition 1.

Corollary 1 *For any behavior P of a given activity specification with event feedback, if Proposition 1 holds for the start and completion times of action and event nodes in P, then for all action and event nodes $n_1, n_2 \in Nod_P$ such that $n_1 \rightarrow n_2$, we have $C'(n_1) \leq S'(n_2)$, meaning that all node dependencies in P are preserved.* ■

By design, the execution engine respects the values of the event outcomes that correspond to the behavior that it executes and processes them in the order specified by the behavior, and it executes all the action and event nodes of that behavior. Therefore, the execution engine preserves the behavior that it executes (Definition 18).

As a result of the reasoning so far, we can formalize the *main contribution of this article* in a theorem that states the time- and behavior preservation of our execution architecture.

Theorem 1 *For any supervisory controller specification (Definition 6) consisting of an activity specification (Definition 3) with a deterministic and complete logistics I/O automaton, under Assumptions 1-6, any behavior that is executed, is executed in a time- (Definition 16) and behavior-preserving (Definition 18) manner.* ■

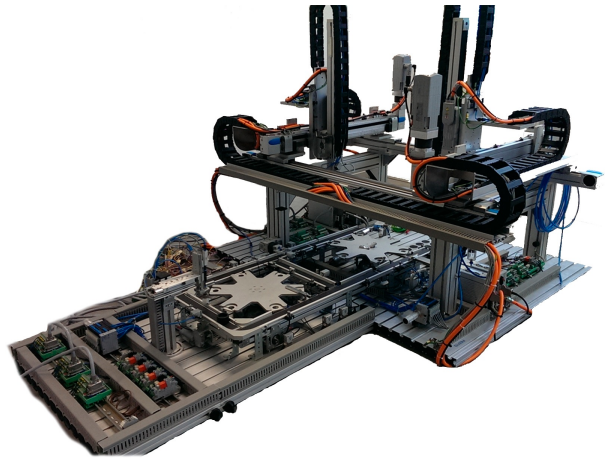
7 | VALIDATION

In this section, we first introduce the eXplore Cyber-Physical Systems (xCPS) [31] platform (Figure 7) as a case study. Then, we validate our approach on it by demonstrating that the execution engine executes the xCPS Supervisory Control (SC) specification in a time- and behavior-preserving manner. The SC specification is an activity model (Definition 3) of the xCPS with a deterministic and complete I/O automaton.

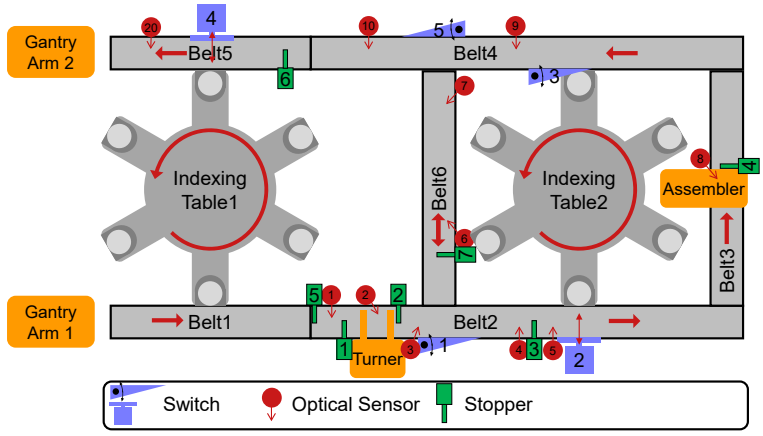
7.1 | Case Study: The xCPS Platform

The xCPS is a small-scale production line. It is capable of producing different products. One of its production scenarios is that it assembles *top* pieces and *bottom* pieces, and outputs the assembled products. The machine captures the features and behavior of FMS present in industry. The pieces go through several processing steps in the machine using belts and stoppers: first, a *gantry arm* collects them and puts them onto an input belt, which moves them to a *turner* station where all tops are correctly oriented. The bottoms are assumed to be correctly oriented so they pass through this station. After the turner station, the bottoms are pushed to an *indexing table* by a *switch*, and the tops move to an *assembler*. The proper routing of pieces is managed through a number of sensors, stoppers, and switches. When a correctly-oriented top arrives at the assembler, the assembler picks it up and waits for the bottom that was pushed to the indexing table. The table rotates and positions the bottom under the assembler station. The assembler then puts the top that it picked on the bottom on the indexing table. Afterward the assembled product is pushed to the output belt by a *switch* and is removed from the assembly line.

We illustrate how our proposed model-driven approach to the realization of supervisory control takes an activity model of the xCPS system that is used to automatically configure the reusable execution architecture to realize xCPS's supervisory control, properly handling the action timing, concurrency, pipelining, synchronization, and event feedback in the system.



(a) The physical layout



(b) Schematic view

FIGURE 7 The xCPS

7.2 | Demonstration

Figure 8 shows a Gantt chart of a behavior of the xCPS producing two products (for brevity), where horizontal bars correspond to actions on peripherals and the arrows in between them are the dependencies. The structure of actions and dependencies is complex, because the actions of the sequences activities are pipelined, and partially concurrent, and synchronizations occur due to resource sharing and processing of event outcomes. Configured with the specification of the xCPS SC, the execution engine becomes the supervisory controller, performing all necessary actions to produce products, while preserving time and behavior for all action and event nodes.

To properly control the xCPS we need to preserve the model start and completion times with timing errors bounded by a time value, denoted by $\mathbb{B}$, determined by time-critical actions of the xCPS. The act of pushing pieces to the indexing table using Switch 2 (see Figure 7) is the most time-sensitive action of the xCPS. This action has a time window of 20 milliseconds to be correctly executed, computed based on belt speed and distance. Missing this time

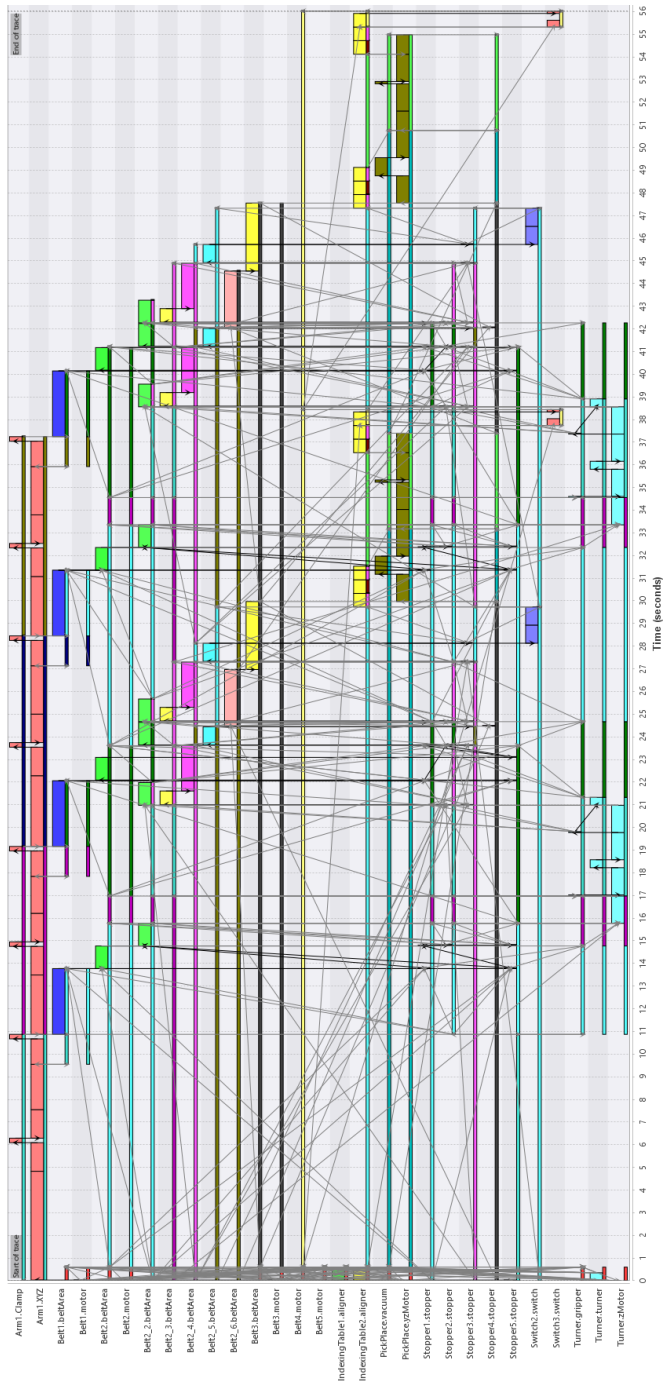


FIGURE 8 Gantt chart of the xCPS producing two products

window means the piece can no longer be pushed to the table (Figure 7b). Therefore, we take $\mathbb{B} = 20$ milliseconds as the required time bound for the execution relation. The execution engine must satisfy this property to ensure proper control of the machine.

Recall that behavior is preserved since, through timing preservation, a node never starts earlier than the completion time of a node it depends on, all nodes of a behavior are executed, and all event outcomes of a behavior are respected (conform Theorem 1).

Given the specification of the xCPS, we must check whether it satisfies Assumptions 5 and 6. Therefore, we first establish the execution delay bound ($\mathbb{D}_{\mathbb{A}}$) of the nodes, as well as the event processing delay bound ($\mathbb{D}_{\mathbb{E}}$). Then, we determine the worst-case execution duration ($\mathcal{T}'(n)$) of actions and the delay of events in the system (Definition 13). With these values, we can determine if the specified duration of action nodes and the specified delay of event nodes, satisfy Assumptions 5 and 6, respectively.

To establish the execution delay bound ($\mathbb{D}_{\mathbb{A}}$), we measure the response time of the aC for every node and check how long the aC takes to process each action or event node. Establishing $\mathbb{D}_{\mathbb{A}}$ is also necessary to check if we can meet the time-criticality of nodes in the xCPS such that $\mathbb{D}_{\mathbb{A}} \leq \mathbb{B}$. Then, from these measurements, we determine $\mathbb{D}_{\mathbb{A}}$. There are variations in these measurements because we run our engine on a Linux machine that has tasks to perform beside the execution engine, and they take time. We observe that the aC takes consistently less than 1.4 milliseconds to process nodes. To allow some margin, we determine $\mathbb{D}_{\mathbb{A}} = 1.6$ milliseconds as the bound that the execution engine guarantees on the execution delay of nodes. Note that establishing $\mathbb{D}_{\mathbb{A}}$ also satisfies Assumption 2.

Next, we establish $\mathbb{D}_{\mathbb{E}}$ based on measurements on the execution engine. Since we run the engine on a Linux machine that cannot guarantee timing, we cannot compute $\mathbb{D}_{\mathbb{E}}$ based on the size of the specification. We use the largest decision path in the specification based on the total number of nodes in its activities to establish the bound $\mathbb{D}_{\mathbb{E}}$. We discussed in Section 6.4 why that is the largest contributor to the time that it takes to process an event. By giving the largest decision path to the engine in a test scenario (not connected to the physical machine), and measuring the time that it takes to sequence all the activities in that decision path, we can determine individual components that make up $\mathbb{D}_{\mathbb{E}}$. Note that the values of $\mathcal{T}(n)$ have no effect on the results of this test. The largest decision path of the xCPS specification has 75 nodes. We observe $\mathbb{D}_{Event} < 0.5$, $\mathbb{D}_{LC} < 1.8$, $\mathbb{D}_{AC} < 2.8$, and $\mathbb{D}_{aC} < 0.8$ milliseconds for the processing time of this decision path. Therefore, we determine $\mathbb{D}_{\mathbb{E}} = 6$ milliseconds (rounded upwards). Further, we observe the time needed to resolve the assembler event is $\mathcal{T}'(e) < 1.9$ milliseconds, which we then use to check the event delays of the model to satisfy Assumption 6. The specified delay of the event node of the assembler is 10 milliseconds which satisfies Assumption 6 since $10 > 1.9 + 6 + 1.6$.

To determine $\mathcal{T}'(n)$ for action nodes, we examined the timings of the actions that the xCPS performs. Some actions, such as movements of the gantry arms, are controlled by a dedicated precision motion controller that moves the arms very accurately. These actions do not need timing measurements since we can determine the bound on their timing based on the configuration of their controller. For actions that have very little variation in their timing, we performed 20 measurements and for the ones that have large variation in their timing, we performed 100 measurements. In both cases, for every action, the bound on their timing was determined to be the maximum over all measurements of that action (Assumption 1).

Having the $\mathcal{T}'(n)$ of all action nodes, $\mathbb{D}_{\mathbb{A}}$, and $\mathbb{D}_{\mathbb{E}}$, we then check their timing in the model. The xCPS specification satisfies Assumption 5 for all nodes.

Our execution engine guarantees time and behavior preservation under Assumptions 1-6. So far, we have determined $\mathbb{D}_{\mathbb{A}}$, $\mathbb{D}_{\mathbb{E}}$, and $\mathcal{T}'(n)$ of all action and event nodes for the xCPS, satisfying Assumptions 1, 2, 5, and 6. Now we check the rest of our assumptions, and then, we execute the complete specification.

To ensure that the I/O automaton of the xCPS has no blocking states (Assumption 3), it was synthesized using the

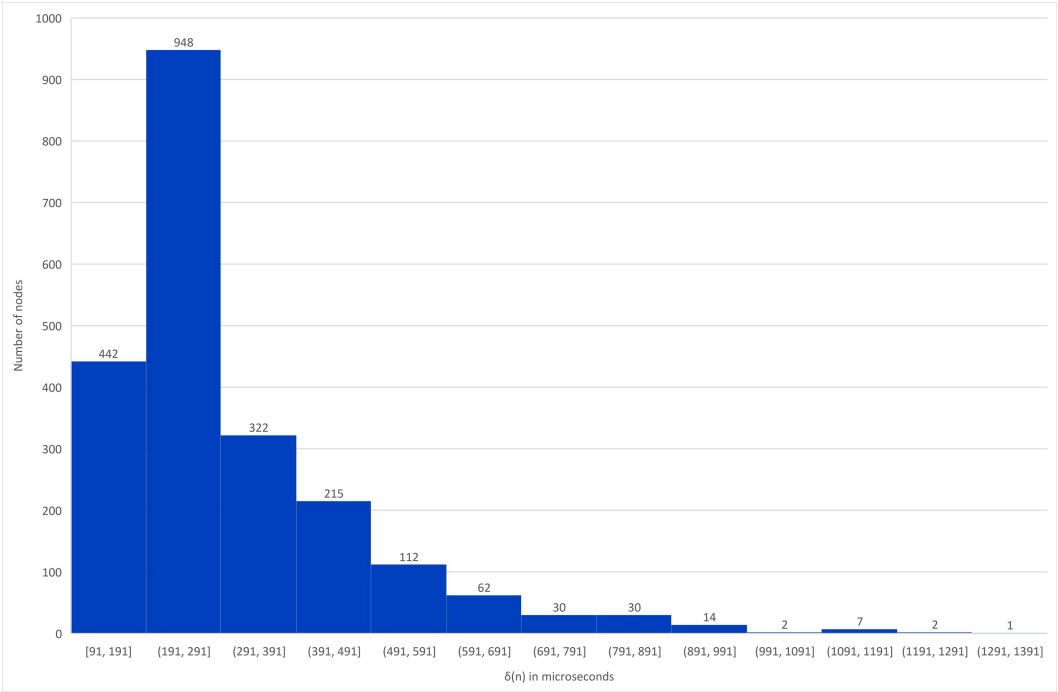


FIGURE 9 $\delta(n)$ for all nodes over 10 runs of the xCPS

approach of [8] with modular automata that describe the order of activities executed in each station of the xCPS. For example, an automaton describes the different allowed orders of activities that can be executed around the indexing table to safely align it with the belts for pushing products on and off of it, as well as aligning it with the assembler to assemble a product. The approach is based on SC synthesis theory of [27], and ensures non-blockingness because transitions leading to blocking states are pruned during synthesis. We synthesized an I/O automaton to produce three products that has 1,169 states and 3,221 transitions. A deterministic I/O automaton was then derived from the result and was given as logistics automaton to the execution engine. This was done by pruning transitions that violated the restrictions of a deterministic I/O automaton. For every state, we only allowed one outgoing transition unless they were based on different outcomes of the same event. Then we removed the states and transitions that became unreachable because of the pruning. As a result of the synthesis, the I/O automaton had only one initial and one final state and no outgoing transition from the final state. The deterministic I/O automaton has 59 states and 61 transitions.

To ensure that the engine has sufficient time to initialize (Assumption 4), we measure the initialization time of the execution engine by initializing the complete specification of the xCPS. Since our implementation is based on a Linux machine, we perform this test 20 times. It consistently takes less than 800 milliseconds to prepare every layer and start executing the nodes. Therefore, we choose Ψ to be the wall clock time one second in the future from when we start the execution.

Now that all the specification and the plant assumptions are satisfied, we run the complete specification 10 times, each time producing three products. Figure 9 shows the distribution of $\delta(n)$ for all action and event nodes over 10 runs of the system. Every run executes 241 action and event nodes, 2410 in total of 10 runs. We observe that

$\mathbb{D}_{\mathbb{A}} \leq \mathbb{B}$. Meaning that the execution engine guarantees an execution delay bound for all nodes that satisfies the time-criticality of actions in the xCPS. We further observe that a great majority (71%) of action and event nodes are executed less than 391 microseconds after their model start time. We notice that $\delta(n)$ increases when there are many nodes starting at the same time and the hardware concurrency of our controller does not provide enough resources to handle all of them concurrently.

Our implementation of the execution engine has been instrumented with OpenTelemetry [32] at the Action Controller and Activity Controller layers. OpenTelemetery is a collection of APIs, SDKs, and tools, used to instrument, generate, collect, and export telemetry data (metrics, logs, and traces) to analyze system performance and behavior. Through this instrumentation any synthesized supervisory controller with the execution engine automatically provides logging information.

In conclusion, this shows that we have delivered the promise of Theorem 1 on a physical FMS. All action and event nodes, including the most time-sensitive ones, are executed within their specified start and completion times, meaning that Proposition 1 is confirmed in a physical setting. The execution engine executes the complete specification of the xCPS. It executes all nodes of a behavior, respects the event outcomes that correspond to the behavior, and preserves node timings and dependencies.

8 | CONCLUSION

In this article, we have introduced an execution architecture and execution engine that allow a specification of system activities to be executed in a way that is guaranteed to be time- and behavior-preserving. We developed a mathematical basis for our execution architecture and engine and captured the time- and behavior preservation that they guarantee in Theorem 1. We proved that the architecture and engine preserve the specified ordering of action and event nodes, as well as their timing according to the model up to a known bound, while respecting the sequence of event outcomes that are received from the plant. Thus, implementations generated by our approach guarantee model-based performance predictions (e.g., with respect to throughput, makespan) as well. We demonstrated our approach on the xCPS prototype production system. The activity framework is a type of activity-based modeling formalism with a strong foundation in max-plus linear algebra that focuses on determinate timing and behavior specifications. With our contribution, we enable a model-driven supervisory controller realization starting from an activity model.

For future work, it would be interesting to investigate how to reduce the $\mathbb{D}_{\mathbb{E}}$ overhead by pipelining the execution of decision paths. Furthermore, the scalability of the framework can be improved by breaking down large decision paths into smaller ones. This allows $\mathbb{D}_{\mathbb{E}}$ to have a ceiling as it linearly grows with the size of decision paths in the model. Finally, it would be interesting to study applications of our activity execution engine (and the accompanying modeling and analysis techniques) in other areas of CPS and to extend the work to richer models of computation such as FSM-SADF and Lingua Franca.

9 | CONFLICTS OF INTEREST

The authors have no conflict of interest to declare that are relevant to this article.

10 | DATA AVAILABILITY STATEMENT

We provide a generic implementation of our execution engine that can be connected to different plants as open source in a publicly available Git repository [33]. Given that the experiment presented in Section 7 was performed on a physical machine, it is not possible to provide software to reconstruct those specific experiments. Instead, the provided version of the software works on a virtual version of the xCPS system that is provided along with the software.

references

[1] Delligatti L. SysML distilled: A brief guide to the systems modeling language. New Jersey: Pearson Education; 2014.

[2] Eriksson HE, Penker M, Lyons B, Fado D. UML 2 toolkit. Indianapolis, Indiana: John Wiley & Sons; 2003.

[3] Schumacher T, Müller CK, Inkermann D. SysML Process Chains in MBSE: Systematic Literature Review and Future Research Directions. Digital Engineering 2025;p. 100037.

[4] Baheti R, Gill H. Cyber-physical systems. The impact of control technology 2011;12(1):161–166.

[5] Opranescu V, Ionita AD. Review of Cyber-Physical Systems Modeling With UML, SysML, and MARTE. IEEE Access 2025;13:47132–47145.

[6] Stecke KE. Formulation and solution of nonlinear integer production planning problems for flexible manufacturing systems. Management science 1983;29(3):273–288.

[7] van der Sanden B, Bastos J, Voeten J, Geilen M, Reniers M, Basten T, et al. Compositional specification of functionality and timing of manufacturing systems. In: 2016 Forum on Specification and Design Languages (FDL) IEEE; 2016. p. 1–8.

[8] van der Sanden B, Reniers M, Geilen M, Basten T, Jacobs J, Voeten J, et al. Modular model-based supervisory controller design for wafer logistics in lithography machines. In: 2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS); 2015. p. 416–425.

[9] van der Sanden B, Blankenstein Y, Schiffelers R, Voeten J. LSAT: Specification and Analysis of Product Logistics in Flexible Manufacturing Systems. In: 2021 IEEE 17th International Conference on Automation Science and Engineering (CASE) IEEE; 2021. p. 1–8.

[10] Nicolescu G, Mosterman P. Model-Based Design for Embedded Systems; 2018.

[11] Schmidt DC. Model-driven engineering. Computer-IEEE Computer Society- 2006;39(2):25.

[12] Mohamadkhani A, Geilen M, Voeten J, Basten T. Modeling and analysis of switching max-plus linear systems with discrete-event feedback. Discrete Event Dynamic Systems 2023;33(3):341–372.

[13] Derler P, Feng TH, Lee EA, Matic S, Patel HD, Zhao Y, et al. PTIDES: A programming model for distributed real-time embedded systems. University of California, Berkeley, EECS Technical Report EECS-2008-72 2008;.

[14] Zou J, Matic S, Lee EA, Feng TH, Derler P. Execution Strategies for PTIDES, a Programming Model for Distributed Embedded Systems. In: 2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium; 2009. p. 77–86.

[15] van Kampenhout R, Stuijk S, Goossens K. A scenario-aware dataflow programming model. In: 2015 Euromicro Conference on Digital System Design IEEE; 2015. p. 25–32.

[16] Zhou MC, Dicesare F, Rudolph DL. Design and implementation of a Petri net based supervisor for a flexible manufacturing system. Automatica 1992;28(6):1199–1208.

[17] Lohstroh M, Menard C, Bateni S, Lee EA. Toward a Lingua Franca for deterministic concurrent systems. ACM Transactions on Embedded Computing Systems (TECS) 2021;20(4):1–27.

[18] Amnell T, Fersman E, Mokrushin L, Pettersson P, Yi W. TIMES: a tool for schedulability analysis and code generation of real-time systems. In: Formal Modeling and Analysis of Timed Systems: First International Workshop, FORMATS 2003, Marseille, France, September 6-7, 2003. Revised Papers 1 Springer; 2004. p. 60–72.

[19] Hui J, Edwards SA. The Sparse Synchronous Model on Real Hardware. ACM Trans Embed Comput Syst 2024 Aug;23(5). https://doi.org/10.1145/3572920.

[20] Benveniste A, Caspi P, Edwards SA, Halbwachs N, Le Guernic P, de Simone R. The synchronous languages 12 years later. Proceedings of the IEEE 2003;91(1):64–83.

[21] Theelen B, Geilen M, Basten T, Voeten J, Gheorghita SV, Stuijk S. A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In: Fourth ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2006. MEMOCODE'06. Proceedings. IEEE; 2006. p. 185–194.

[22] Peterson JL. Petri net theory and the modeling of systems. 1 ed. Upper Saddle River, NJ, United States: Prentice Hall PTR; 1981.

[23] Alur R, Dill DL. A theory of timed automata. Theoretical computer science 1994;126(2):183–235.

[24] Jerad C, Lee EA. Toward Dynamism in Distributed Lingua Franca Programs. IEEE Embed Syst Lett 2024 Sep;17(2):111–114. https://doi.org/10.1109/LES.2024.3465408.

[25] Paladino F, Jellum E, Soyer E, Lee EA. Layered Scheduling: Toward Better Real-Time Lingua Franca. IEEE Embed Syst Lett 2024 Sep;17(2):127–130. https://doi.org/10.1109/LES.2024.3467992.

[26] Kopetz H, Bauer G. The time-triggered architecture. Proceedings of the IEEE 2003;91(1):112–126.

[27] Ramadge PJ, Wonham WM. Supervisory control of a class of discrete event processes. SIAM journal on control and optimization 1987;25(1):206–230.

[28] Digitale JC, Martin JN, Glymour MM. Tutorial on directed acyclic graphs. Journal of Clinical Epidemiology 2022;142:264–267.

[29] Hu B, Xu S, Cao Z, Zhou M. Safety-Guaranteed and Development Cost- Minimized Scheduling of DAG Functionality in an Automotive System. IEEE Transactions on Intelligent Transportation Systems 2022;23(4):3074–3086.

[30] Lynch NA, Tuttle MR. An introduction to input/output automata. Massachusetts, United States: Laboratory for Computer Science, Massachusetts Institute of Technology; 1988.

[31] Adyanthaya S, Ara HA, Bastos J, Behrouzian A, Sánchez RM, van Pinxten J, et al. xCPS: a tool to explore cyber physical systems. ACM SIGBED Review 2017;14(1):81–95.

[32] OpenTelemetry;. Accessed: 2024-12-16. https://opentelemetry.io/.

[33] Git repository of the Activity Execution Engine;. Accessed: 2025-08-25. https://github.com/TUE-EE-ES/activity-execution-engine.