Execution Strategies for PTIDES, a Programming Model for Distributed Embedded Systems

Jia Zou, Slobodan Matic, Edward A. Lee, Thomas Huining Feng
Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, USA.
{jiazou, matic, eal, tfeng}@eecs.berkeley.edu

Patricia Derler
Computer Science
University of Salzburg
Salzburg, Austria
Patricia.Derler@cs.uni-salzburg.at

***Abstract*—We define a family of execution policies for a programming model called *PTIDES* (Programming Temporally Integrated Distributed Embedded Systems). A PTIDES application (factory automation, for example) is given as a discrete-event (DE) model of a distributed real-time system that includes sensors and actuators. The time stamps of DE events are bound to physical time at the sensors and actuators, turning the DE model into an executable specification of the system with explicit real-time constraints. This paper first defines a general execution strategy that conforms to the DE semantics, and then specializes this strategy to give practical, implementable and distributed policies. Our policies leverage network time synchronization to eliminate the need for null messages, allow independent events to be processed out of time stamp order, thus increasing concurrency and making more models feasible (w.r.t. real-time constraints), and improve fault isolation in distributed systems. The policies are given in terms of a *safe to process* predicate on events that depends on the time stamp of the events and the local notion of physical time. In a simple case we show how to statically check whether program execution satisfies timing constraints.**
[1]

I. Introduction

This paper considers distributed real-time embedded systems such as factory automation, large-scale instrumentation systems, and network supervision and control. Implementations of such systems consist of networked computers ("platforms") with sensors and actuators distributed throughout a network. Orchestrated actions are required from the platforms.

We assume that the application is described using the PTIDES model introduced in [18]. PTIDES is an acronym for *programming temporally integrated distributed embedded systems*. In PTIDES, rather than defining software tasks as threads with periods, priorities, and deadlines, as is typically done for embedded systems, an application is given as a model in a discrete-event (DE) modeling language. Whereas classically DE would be used to construct *simulations* of such systems, in PTIDES, the DE model is an executable specification. The objective is to compile this specification into a deployable implementation. Thus, PTIDES follows the principles of model-based design [11].

PTIDES builds on the solid semantic foundation of DE models [12], which makes it much easier to get determinate concurrent composition of software components than it is with threads [13]. Moreover, a key strength of PTIDES is that its distributed software specifications are explicit about end-to-end latency between sensors and actuators, making the behavior of the software in the context of cyber-physical systems much more repeatable and predictable. This contrasts with the more indirect mechanisms typically used, where for example priorities for software tasks (vs. actuator actions) are empirically determined and experimentally verified.

Distributed DE simulation is an old topic [7]. The focus has been on accelerating simulation by exploiting parallel computing resources. A brute-force technique for distributed DE execution uses a single global event queue that sorts events by time stamp. This technique, however, is only suitable for extremely coarse grained computations, and it provides a vulnerable single point of failure. For these reasons, the community has developed distributed schedulers that can react to time-stamped events concurrently. So-called "conservative" techniques process time-stamped events only when it is known to be safe to do so [4]. It is safe to process a time-stamped event if we can be sure that at no time later in the execution will an event with an earlier time stamp appear that should have been processed first. So-called "optimistic" techniques [10] speculatively process events even when there is no such assurance, and roll back if necessary.

For distributed embedded systems, the potential for roll back is limited by actuators (which cannot be rolled back once they have had an effect on the physical world) [6]. Established conservative techniques, however, also prove inadequate. In the classic Chandy and Misra technique [4], each compute platform in a distributed simulator sends messages even when there are no events to convey in order to provide lower bounds on the time stamps of future messages. This technique carries an unacceptably high price in our context. In particular, messages need to be frequent enough to prevent violating real-time constraints due to waiting for such messages. Messages that only carry time stamp information and no data are called "null messages." Not only that these messages increase networking overhead, but the technique is also not robust. Failure of single component results in no more such messages, thus blocking

[1]This work was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF awards #0720882 (CSR-EHS: PRET), #0647591 (CSR-SGER), and #0720841 (CSR-CPS)), the U. S. Army Research Office (ARO #W911NF-07-2-0019), the U. S. Air Force Office of Scientific Research (MURI #FA9550-06-0312), the U. S. Air Force Office of Scientific Research (AF-TRUST #FA9550-06-1-0244) the Air Force Research Lab (AFRL), the State of California Micro Program, and the following companies: Agilent, Bosch, HSBC, Lockheed-Martin, National Instruments, and Toyota.

progress in other components. Our work is related to several efforts to reduce the number of null messages, such as [7], but makes much heavier use of static analysis.

[18] leverages static analysis of DE models to achieve distributed DE scheduling that is conservative but does not require null messages. The static analysis enables independent events to be processed out of time stamp order. For events where there are dependencies, the technique goes a step further by requiring clocks on the distributed computational platforms to be synchronized with bounded error. Recently, techniques such as IEEE 1588 [8] have been developed to deliver synchronization precision of nanosecond order over a local area network. This enables truly game-changing opportunities for distributed embedded software. In our case, the mere passage of time obviates the need for null messages.

Considerable research activity has been devoted to exploring similarly high-level MoCs for embedded systems. The classic SDL model defines embedded systems as asynchronously communicating processes, and has been extended with models of time [16]. FunState uses a functional programming style driven by state machines for imperative logic [17]. The synchronous languages Esterel, Lustre, Signal, SCADE, and various dialects of Statecharts have long been explored for the specification and design of embedded systems [3]. BIP gives embedded systems as interacting state machines [2]. ForSyDe [9] provides disciplined mixtures of MoCs for embedded systems design. And there are many others. The model we discuss here is unique (we believe) in building on classical discrete-event techniques long used for simulation [7].

In this paper, we define a family of execution strategies for PTIDES that assure compliance with DE semantics and support distributed scheduling and control (Sec. II). We then specialize this general strategy to give practical, implementable distributed policies with varying cost-performance trade-offs (Sec. III). Here we improve the strategy from [18] using the concept of a dependency cut. Our policies: 1) leverage network time synchronization to eliminate the need for null messages, 2) improve fault isolation in distributed systems by making it impossible for components to block others by failing to send messages, and 3) allow independent events to be processed out of time stamp order, thus increasing concurrency and making more models feasible (w.r.t. real-time constraints). The policies are given in terms of a *safe to process* predicate on events that depends on the time stamp of the events and a local notion of physical time. We discuss the trade-offs of the policies through the use of a manuafacturing assembly line example (Sec. IV). Finally, for a simple setting we provide a feasibility analysis w.r.t. the real-time constraints of the system (Sec. V).

II. PTIDES Execution Strategy

We leverage static dependency information between actors to develop an execution strategy for discrete-event models. This strategy is general in the sense that it allows for different implementations targeting a variety of computer architectures. An implementation and a time-synchronized architecture making use of this strategy are discussed in the next section.

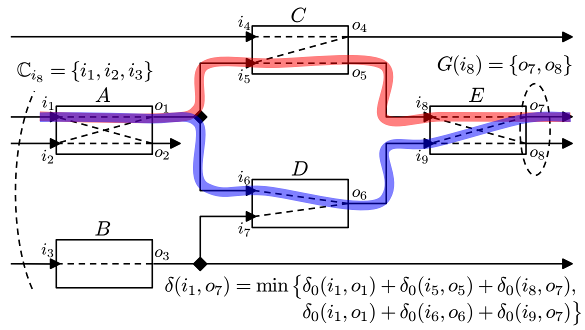
$\mathbb{C}_{i_8} = \{i_1, i_2, i_3\}$
$G(i_8) = \{o_7, o_8\}$
A
B
C
D
E
i_1
i_2
i_3
i_4
i_5
i_6
i_7
i_8
i_9
o_1
o_2
o_3
o_4
o_5
o_6
o_7
o_8
$\delta(i_1, o_7) = \min\{\delta_0(i_1,o_1) + \delta_0(i_5,o_5) + \delta_0(i_8,o_7),$
$\delta_0(i_1,o_1) + \delta_0(i_6,o_6) + \delta_0(i_9,o_7)\}$

Fig. 1. Example with minimum model-time delay and dependency cut.

A. Model Structure and Dependencies

We specify DE models using the actor-oriented approach [5]. Actors are concurrent components that exchange time-stamped events via input and output ports. Actors react to input messages by “firing,” by which we mean performing a finite computation and possibly sending output messages. The “time” in time stamps is *model time*, not *wall clock time*, which we also call *physical time* in this paper. DE semantics requires that each actor processes input events in time-stamp order. It does not impose any constraints on the physical time at which events are processed. In this paper we will assume that time stamps are non-negative real numbers. However, to fully support heterogeneous modeling and design the formalism should be extended with super-dense time [14].

We assume a port to be either an input port or an output port. This is without loss of generality, because a port that is an input port and an output port at the same time can be modeled as two distinct ports. We further assume ports to be interconnected by a fixed and static network, where at most one output port is connected to each input port.

A model in our formal representation consists of a set of actors, represented by $\mathcal{A}$. Any actor $\alpha \in \mathcal{A}$ has a set of input ports I_α and a set of output ports O_α. The set of all input ports is $I = \bigcup_{\alpha \in \mathcal{A}} I_\alpha$. The set of all output ports is $O = \bigcup_{\alpha \in \mathcal{A}} O_\alpha$. The set of all ports is $P = I \cup O$.

We represent the input-output dependency between ports with *minimum model-time delay*, which is computed statically. It is formulated as a causality interface [19] using min-plus algebra [1]. We require a function $\delta_0 \colon P \times P \to \mathbb{R}^+ \cup \{\infty\}$ to be provided *a priori*, where $\mathbb{R}^+$ is the set of non-negative real numbers. By requiring the return values be non-negative, we explicitly assume the actors we are dealing with to be *causal*, in the sense that their output events are no earlier in model time than the input events that cause them.

The function δ_0 is defined as follows.

1) If p_1 is an output port, p_2 is an input port, and p_1 is connected to p_2, then $\delta_0(p_1, p_2) = 0$.
2) If $p_1 \in I_\alpha$ and $p_2 \in O_\alpha$ for some $\alpha \in \mathcal{A}$, then $\delta_0(p_1, p_2)$ is provided by the designer of actor α to characterize the dependency between input port p_1 and output port p_2. Alternatively, it may be inferred from a hierarchical definition of α using the methods of [19]. In either case, if $\delta_0(p_1, p_2) = \tau_0$ (where $\tau_0 \in \mathbb{R}^+$), the actor guarantees that an input event at p_1 with time stamp τ has no effect on any event(s) at p_2 with time stamp less than $\tau + \tau_0$.

3) For all other ports p_1 and p_2, $\delta_0(p_1, p_2) = \infty$.

For example, for a *Delay* actor with input port p_1, output port p_2 and a constant model-time delay τ_D between them ($\tau_D \geq 0$), $\delta_0(p_1, p_2) = \tau_D$. For a *VariableDelay* actor, whose delay can be changed at run-time but is always non-negative, $\delta_0(p_1, p_2) = 0$. If the events at an input port p_1 never affect those at an output port p_2, then $\delta_0(p_1, p_2) = \infty$.

We use the model in Fig. 1 as a running example to clarify the definitions in this section. In that figure, rectangles represent actors, while filled triangles pointing into and going out of actors are inputs and output ports, respectively. The input ports are labeled i_1 through i_9 and the output ports are labeled o_1 through o_8. A dashed line in an actor represents predefined non-infinity dependency between the connected input port and output port. For example, the dashed line between i_1 and o_1 in actor A implies that $\delta_0(i_1, o_1)$ is statically known to be a number in $\mathbb{R}^+$.

A *path* from port p_1 to p_n is a sequence of ports $[p_1, p_2, \cdots, p_n]$ for some $n > 0$. A *subpath* is a sequence of consecutive ports in a path. In Fig. 1, $[i_1, o_1, i_5, o_5, i_8, o_7]$ is a path, and $[i_5, o_5, i_8]$ is a subpath. We define $\delta_P(\rho)$ for path $\rho = [p_1, p_2, \cdots, p_n]$ as the model-time delay on the path as follows. If $n = 1$, then $\delta_P(\rho) = 0$. Otherwise,

$$\delta_P(\rho) = \sum_{k=1}^{n-1} \delta_0(p_k, p_{k+1}).$$

To continue with the previous example in Fig. 1, $\delta_P([i_1, o_1, i_5, o_5, i_8, o_7]) = \delta_0(i_1, o_1) + \delta_0(i_5, o_5) + \delta_0(i_8, o_7)$, where we observe that $\delta_0(o_1, i_5) = \delta(o_5, i_8) = 0$.

Now we are ready to define the minimum model-time delay for arbitrary pairs of ports with function $\delta : P \times P \to \mathbb{R}^+ \cup \{\infty\}$. For any $p_x, p_y \in P$, $\delta(p_x, p_y)$ is defined as follows,

$$\delta(p_x, p_y) = \min\left\{\delta_P(\rho) \mid \rho \text{ is a path from } p_x \text{ to } p_y\right\}$$

That is, $\delta(p_x, p_y)$ is the smallest model-time delay on any path from p_x to p_y. Since in this paper model time is represented with real numbers, we require no cycles with zero model-time delay. In Fig. 1, there are only two paths from i_1 to o_7 that may not yield ∞, so the minimum model-time delay from i_1 to o_7 is:

$$\begin{aligned}\delta(i_1, o_7) &= \min\big\{\delta_P([i_1, o_1, i_5, o_5, i_8, o_7]),\\ &\qquad \delta_P([i_1, o_1, i_6, o_6, i_9, o_7])\big\}\\ &= \min\big\{\delta_0(i_1, o_1) + \delta_0(i_5, o_5) + \delta_0(i_8, o_7),\\ &\qquad \delta_0(i_1, o_1) + \delta_0(i_6, o_6) + \delta_0(i_9, o_7)\big\}.\end{aligned}$$

The minimum model-time delay function δ can be computed in a static analysis before execution.

B. General Execution Strategy

In this section, we discuss our execution strategy for distributed discrete-event systems. It is general enough to serve as the basis of a variety of concrete execution policies. For ease of discussion, we make an assumption that, conceptually, an input queue is maintained for each input port. An actor removes events from its input queues only when those events are processed, and the generated output events (if any) are delivered to the input queues of the receiving ports. Optimization is possible by employing fewer event queues, as is done in an implementation described in the next section.

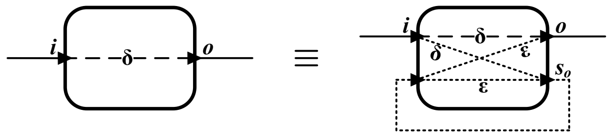

Fig. 2. Actor state and causality interface.

The core of our execution strategy is to decide whether it is safe to process an input event. We assume statetful actors, i.e., the output of event processing depends on the current state of the actor. For an actor shown on the left side of Fig. 2, let i be an input and o an output port such that $\delta_0(i, o) = \delta$ (with $\delta \in \mathbb{R}^+$) and let s be the state of the actor. Thus, if processing of an event with time stamp $\tau - \delta$ at input port i resuts in the event with time stamp τ at output port o we have $o(\tau) = f_o(i(\tau - \delta), s(\tau - \epsilon))$ where f_o is the output function for port o and $\tau - \epsilon$ is the time at which state s has been modified last. To simplify the presentation of event processing in Sec. III-D we here assume that both output and state computation are performed in a single step. Thus, we implicitly assume that for *each* port o of an actor there exists a state port s_o and for *each* input port i such that $\delta_0(i, o) = \delta$ there is also dependency between i and s_o such that $\delta_0(i, s_o) = \delta$. This means that the state will be updated according to $s_o(\tau) = f_{s_o}(i(\tau - \delta), s_o(\tau - \epsilon))$ where f_{s_o} is the state function for state s_o. Therefore, whenever in our figures we show an actor similar to the one shown on left in Fig. 2 we implicitly assume the structure shown on right in the same figure. This assumption can be avoided in a more general formalism that requires super-dense time and fixed-point computation [14]. The issues present in such a formalism are orthogonal to the details of the execution strategy which is the focus of this paper, so in the rest of the paper we assume that the assumption holds.

Consequently, an event at an input port is *safe to process* and produce event with time stamp τ at an output port o when all events at o with time stamps smaller than τ have already been produced. Consider, for instance, actor *D* in Fig. 1. Let time stamps of input events e_6 and e_7 at input ports i_6 and i_7 be τ_6 and τ_7, respectively, and let $\tau_6 < \tau_7$. The two input events may result in output events e'_6 and e'_7 with timestamps $\tau_6 + \delta_0(i_6, o_6)$ and $\tau_7 + \delta_0(i_7, o_7)$. If $\tau_6 + \delta_0(i_6, o_6) > \tau_7 + \delta_0(i_7, o_7)$, then we would process event e_7 first, because it produces an event of smaller time stamp at the output. Notice that this implies all actors produce output events in time stamp order. Of course, if $\delta_0(i_6, o_6) = \delta_0(i_7, o_7)$ events at input ports i_6 and i_7 will be processed in time stamp order. This analysis leads us to extend the notion of dependency by considering a group of output ports that are affected by the same input port.

We define function $G : I \to 2^O$ to return a *group of ports* of the same actor that we need to consider before processing an event at a given port. For any $i \in I_\alpha$,

$$G(i) = \Big\{o \in O_\alpha \mid \delta_0(i, o) < \infty\Big\}.$$

For example, in Fig. 1, $G(i_8) = \{o_7, o_8\}$.

A set $\mathbb{C}_i \subseteq I$ is called a *dependency cut* for input port $i \in I$ if it is a minimal set of input ports that satisfies the following condition.

For any $o_y \in G(i)$ and any path ρ to o_y satisfying $\delta_P(\rho) < \infty$, there exist input port $i_x \in \mathbb{C}_i$ and path ρ' from i_x to o_y satisfying $\delta_P(\rho') < \infty$, such that either ρ is a subpath of ρ' or ρ' is a subpath of ρ.

Intuitively, a dependency cut for i is a "complete" set of ports on which ports in $G(i)$ depend. Completeness in this case means that for each port in $G(i)$, all ports it depends on will be accounted for in $\mathbb{C}_i$, either directly by being included or indirectly by having either upstream or downstream ports included. The dependency cut for a given input port is not unique. Again using Fig. 1 as an example, the dashed curve depicts one possible dependency cut for i_8, namely $\mathbb{C}_{i_8} = \{i_1, i_2, i_3\}$.

We now use the definition of dependency cut to define our general execution strategy:

Given a dependency cut $\mathbb{C}_i$ for input port i of actor α, an event e at i with time stamp τ is safe to process if for any $i_x \in \mathbb{C}_i$ and any $o_y \in G(i)$,

1) *i_x has received all events with time stamps less than or equal to $\tau - \delta(i_x, o_y) + \delta_0(i, o_y)$, and*
2) *for any $i_z \in I$ such that $\delta(i_x, i_z) < \infty$,*
    a) *if $i_z \in I_\alpha$, all events in input queue of i_z have time stamps greater than or equal to $\tau - \delta_0(i_z, o_y) + \delta_0(i, o_y)$,*
    b) *if $i_z \notin I_\alpha$, all events in input queue of i_z have time stamps greater than $\tau - \delta(i_z, o_y) + \delta_0(i, o_y)$*

Recall I_α is the set of input ports for actor α. Intuitively, these conditions ensure that actor α has received all events that can possibly invalidate the processing of e. The first condition ensures that no future events will be received at the ports in the dependency cut $\mathbb{C}_i$ that can possibly affect an output in $G(i)$ earlier. The second condition ensures that no event at the ports between any port in $\mathbb{C}_i$ and any port in I_α can possibly affect an output in $G(i)$ earlier. Notice that if $\delta(i_z, o_y) = \infty$, then this condition is trivially satisfied.

The above execution strategy is general in the sense that it describes only a principle that needs to be satisfied for correctness, while allowing designers to create a spectrum of concrete implementations based on that principle. It relaxes the DE execution policy considerably, clarifying that we only need to know whether an event is "oldest" among the events that can appear in a dependency cut. We do not need to know that it is globally oldest. Of course, the choice of dependency cuts will have a significant effect on how much this relaxes the scheduling.

The distributed execution policy developed by Chandy and Misra [4] also satisfies the principle. That policy can be considered as a special case in which the dependency cut $\mathbb{C}_i$ for input port $i \in I$ is always chosen as I_α. Another policy, which is introduced in [18] and will be elaborated with improvements in the next section, is also a specialization of the principle. That policy takes advantage of the synchronized clocks between distributed platforms.

III. PTIDES Implementation

Since we are focused on distributed embedded systems rather than distributed simulation, some of the actors are wrappers for sensors and actuators. Sensors and actuators interact with the physical world, and we can assume that in the physical world, there is also a notion of time. An actor that wraps a sensor or an actuator cannot produce or consume time-stamp events at arbitrary times. We assume time stamps of events emerging from sensor actors represent the physical time at which a physical measurement was taken. We also assume that the time stamp of an event delivered to an actuator actor represents the physical time at which we wish the actuator to take action.

Along with sensors and actuators, we will also impose timing constraints at network interfaces. A network connection carries time-stamped events from one compute platform to another, and we abstract such an interface as an actor with one input port and one output port. Here, we will assume a bounded network delay, so that the physical time that elapses between delivery of such an event to the network interface and appearance of the event at the output of the network interface is bounded. We also assume that each compute platform in a distributed system maintains a clock that measures the passage of physical time, and any two clocks in the system agree on the notion of physical time up to some bounded error.

At actors that are neither sensors, actuators or network interfaces, there is no relationship between physical and model time. At these actors, output events must be produced in model-time order, but this production can occur at any physical time (earlier or later than the time stamp).

During the execution of distributed DE models, time-stamped events can arrive unpredictably over the network. If the model includes sensor actors that can produce events at arbitrary times, then a similar problem occurs. In this section, we specialize the general execution strategy from Sec. II-B to handle these situations. We use the notion of *real-time ports*, which are ports where time stamps have a particular defined relationship to physical time.

A. Real-Time Ports

As in Sec. II-B, we assume that each input port maintains a queue of as-yet unprocessed events. An input port that is a real-time port has the constraint that at any physical time t, for each event e in the queue,

$$\tau \geq t, \tag{1}$$

where τ is the time stamp of event e. The input ports of actuator actors and network interface actors are normally such real-time ports.

This constraint imposes a physical time deadline on delivery of each event to the queue, because if the event is delivered at a physical time $t > \tau$, then upon delivery, there will be an event in the event queue that violates the constraint. Moreover, this requirement imposes a deadline on the processing of the event, because if the actor is not fired prior to physical time $t = \tau$, then the event will remain on the queue past the point where it satisfies the constraint.

An output port o that is a real-time port has the constraint that if it produces an event e with time stamp τ at physical time t, then

$$\tau + d_o \geq t \tag{2}$$

where $d_o \in \mathbb{R}^+ \cup \{\infty\}$ is a parameter of the port called its *maximum physical time delay*. Here, what we mean by "producing an event" is delivering it to the input queue of all destination input ports.

Output ports of sensor and network interface actors are normally real-time ports. For a sensor, the time stamp of an output event represents the time at which the reported measurement was taken. The constraint $\tau + d_o \geq t$ asserts that the sensor does such reporting in bounded time, if $d_o < \infty$. In case of sensor real-time output ports, it also holds $t \geq \tau$, since sensors can only report about past properties of the physical environment, not future properties.

A *network interface* actor abstracts a network connection, and has a single input port called *network input port*, a single output port called *network output port*, both of which are real-time ports. In this case, d_o is a bound on network latency plus the network synchronization error. We make a special introduction of this actor here because this is the only actor that is used for different platforms to communicate between each other within a distributed system.

Note that Eq. (2) represents a timing assumption on system inputs, while Eq. (1) represents the timing guarantee on system outputs. Whether that guarantee can be met in every execution is the *schedulability* question, which we address only in a simple case through our feasibility analysis in Sec. V.

B. Examples

We use the example in Fig. 3 to introduce initial events. First we look at the *Source* actor in this example. At each firing of this actor, it consumes one event and produces two events, one at each output port. At the start of execution, an initial event is inserted into the input port of this actor. In figures we indicate such events with blue dots. Since all actors other than sensors can only produce events after the consumption of some other event, these initial events are key to start the execution of the model. Note that these initial events do not have to bear any relationship between model time and physical time. Also note if there exists an actor which spontaneously produces events without consuming any event, that actor can be modeled by the *Source* actor as shown in our example. In addition, if our model has a loop of actors, we also need at least one initial event to start it.

To show the selection of a suitable dependency cut, we consider the same example. Two real-time ports exist in Fig. 3: output port o of the *Sensor* actor and the input port i of the *Actuator* actor. Suppose we have received an event e with time stamp τ at i_2. When is it safe to process e (i.e., to fire the *Computation* actor)? Following the general execution policy, we need to first choose a dependency cut $\mathbb{C}_{i_2}$. i_1, i_3 is one option for the cut. Notice that i_1 relates model time to physical time, and we chose i_3 over i_2 because an initial event resides at this port, which indicates it is a "source" of events. With $\mathbb{C}_{i_2} = \{i_1, i_3\}$, the general execution policy tells us that we can process the event if i_1 and i_3 have received all events with time stamp less than or equal to τ. Assume for the *Source* actor $\delta_1 = \delta_2$. Then on i_3 this requirement is automatically true. On i_1, because of the constraint (2) in Sec. III-A on o, the requirement is guaranteed to have happened when physical time t is greater than $\tau + d_o$.

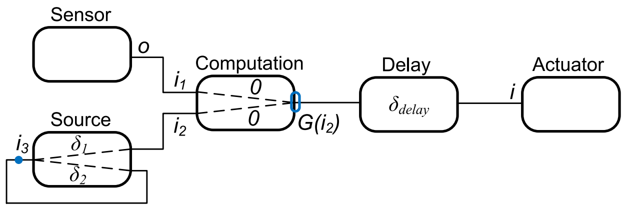

Fig. 3. Simple DE model with a sensor and an actuator.

C. Dependency Cut Selection

Motivated by the above examples, we provide an algorithm for choosing a suitable dependency cut C_i for a given input port i.

Since the dependency cut specifies how far upstream we need to look to determine whether an event is safe to process, it is important to make the cut such that events that occur within the cut are easily accessible; this simplifies the safe-to-process analysis. Given that, it makes sense for us to find a dependency cut within the same platform as i. Further, we observe that if the dependency cut consists of ports that relate model time to physical time, then we can completely eliminate the need for null messages across computing platforms. Thus our goal is to define the cut at the "boundary" of the platform, where "boundary" implies we make the cut at ports that communicate with the outside of the platform, which may be the rest of the distributed system or the physical environment.

Note here our choice of the cut is determined by the assumption that communication within the platform is economical, while communication outside of the platform is expensive. But this may not be true in all cases.

Fig. 4 shows an example of the cut C_{i_n} of port i_n for each $n \in \{1, 2, 3\}$ (notice that the cuts for all these ports are identical). Ports i'_1 and i'_4 are candidates because they are input ports connected to real-time output ports, i'_5 is a candidate because it is a real-time input port, while i'_2 and i'_3 are candidates because initial events are present at the start of execution at these ports. Now to ensure the cut is minimal, we see that i'_4, i'_5 can be reached by traversing the graph from i'_3, thus they are removed from the cut. Hence we have $C_{i_n} = \{i'_1, i'_2, i'_3\}$ for each $n \in \{1, 2, 3\}$.

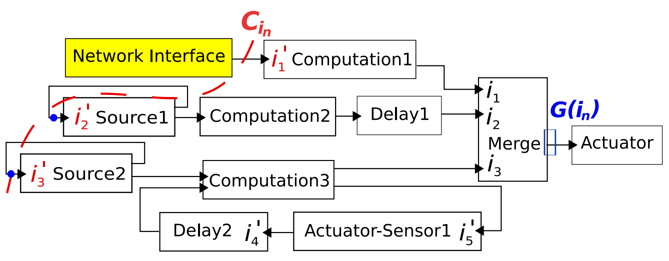

Fig. 4. Example: dependency cut for $G(i_n)$.

Using the above example as motivation, we formally describe the cut selection procedure using standard graph notation and algorithms. Let us define a directed graph $G =$

(V, E, W, L) that describes the PTIDES model being examined, where

$$\begin{aligned}
&V = P, \\
&E = \{(v_1, v_2) \mid \forall v_1, v_2 \in V.\ \delta_0(v_1, v_2) < \infty \\
&\quad \wedge v_1 \text{ is not a network input port} \\
&\quad \wedge v_2 \text{ is not a network output port}\}, \\
&W(v_1, v_2) = \delta_0(v_1, v_2), \text{and}
\end{aligned}$$

$$L(v) = \begin{cases} 1 & \text{if } ((v', v) \in E \wedge v' \text{ is real-time output port}) \\ & \vee\ v \text{ is a real-time input port} \\ & \vee\ v \text{ is an input port with initial events}, \\ 0 & \text{otherwise.} \end{cases}$$

Here, V is the set of ports of the model; E is the set of edges; $W : E \to R^+ \cup \{\infty\}$ is a weight function that maps each edge to its minimal model time delay; $L : V \to \{0, 1\}$ is a labeling function that determines whether each vertex $v \in V$ is a candidate for dependency cut. This graph constructs the original model, while disconnecting the edges from network input ports or towards network output ports. Since using network interface actors is the only way data could be communicated across platforms, this in effect removes all connections across computation platforms. This ensures the cut always consists of the ports from the same platform as port group $G(i)$.

We determine a dependency cut $\mathbb{C}_i$ for port i in a two-step algorithm as follows:

1) *for each $v \in V$ such that $L(v) = 1$, start from v and traverse G. If the traversal leads to a vertex $i' \in G(i)$, add v to $\mathbb{C}_i$.*
2) *After step 1 is completed, start from each $v \in \mathbb{C}_i$ and traverse G. If during this process a vertex v' is reached such that $v' \in \mathbb{C}_i$, update $\mathbb{C}_i$ by removing v' from the cut.*

Notice that step 1 of our algorithm ensures the cut is complete. Step 2 of our algorithm guarantees the cut is minimal, since the traversal ensures any two vertices belonging to a path will not be part of the cut at the same time. Also, in the case where we have more than one candidate for the cut within a cycle, the above algorithm is not deterministic in specifying which one among them will become a member of the cut. However, we do not care which port is chosen through step 2 of the algorithm, as long as exactly one among them is chosen, and step 2 ensures exactly that.

Also notice that if our only goal of the cut selection algorithm is to find the dependency cut, then any graph traversal algorithm could be used in step 1. However, during traversal, it is also beneficial for us to obtain the value of minimum model time delay δ from each member of the cut to each element of $G(i)$. This value could be obtained by using a shortest path algorithm as a graph traversal algorithm.

D. Safe-to-Process Analysis

Using the dependency cut obtained by the algorithm in Sec. III-C, in this subsection we present an instance of the general execution strategy described in Sec. II-B, i.e., we present conditions for safe processing of events that obeys the DE semantics.

We first define the *physical-time delay* function $d\colon I \to \mathbb{R}^+ \cup \{-\infty\}$ that maps each port $p \in I$ as follows:

$$d(p) = \begin{cases} 0 & \text{if } p \text{ is a real-time input port (1)}, \\ d_o & \text{if } p \text{ is a non-real-time input port} \\ & \quad \text{connected to a real-time output port (2)}, \\ -\infty & \text{otherwise (3).} \end{cases}$$

Here d_o is the maximum physical-time delay specific to the real-time output port p, as defined in equation (2) of Sec. III-A (i.e., $\tau + d_o \geq t$). Note also that equation (1) in the same section can be rewritten as $\tau + 0 \geq t$. Thus, for each input port p that satisfies condition (1) or (2) of the definition $d(p)$ given above we have that an event with time stamp τ is delivered to the input queue of p at physical time t such that $\tau + d(p) \geq t$. For all other ports, i.e., for case (3) above, $d(p) = -\infty$ because no such constraint between physical time and model time exists.

Assume that the model-time delay function δ and physical-time delay function d are given and that for each input port $i \in I$ the dependency cut $\mathbb{C}_i$ is determined according to the algorithm from Sec. III-C. In addition assume $G(i)$ is nonempty. In that case, a procedure for determining safe-to-process events based on the general execution strategy presented in Sec. II-B can be given as follows:

Strategy A. *An event at input port $i \in I$ with time stamp τ is safe to process when:*

1)
$$\tau + \max_{p \in \mathbb{C}_i, o \in G(i)} \{d(p) - \delta(p, o) + \delta_0(i, o)\},$$
and

2) *for each port $p' \in I$ such that there exists port $p \in \mathbb{C}_i$ with $\delta(p, p') < \infty$, each event in input queue of p' has time stamp*
   a) *greater than or equal to* $\tau + \max_{o \in G(i)}\{-\delta_0(p', o) + \delta_0(i, o)\}$ *for all* $o \in G(i)$ *such that* $\delta_0(p', o) < \infty$,
   b) *greater than* $\tau + \max_{o \in G(i)}\{-\delta(p', o) + \delta_0(i, o)\}$ *for all* $o \in G(i)$ *such that* $\delta_0(p', o) = \infty$.

The conditions 1) and 2) correspond to the conditions 1) and 2) of the general execution strategy respectively. The forms of the corresponding conditions 2) are similar. As in the general strategy, condition 2a) checks among events at the ports of the same actor ($\delta_0(p', o) < \infty$), whereas 2b) checks the rest ($\delta_0(p', o) = \infty$). However, condition 1) differs because the intention here is to take advantage of the particular dependency cut $\mathbb{C}_i$. If $d(p) > -\infty$ for an input port $p \in I$, then constraint $\tau + d(p) \geq t$ between physical time and model time can be exploited as explained above. In addition, this condition takes into account all ports of $\mathbb{C}_i$ and $G(i)$, model-time delay δ between those and model-time delay δ_0 between i and ports of $G(i)$. Thus, after the specified physical time the event can be safely processed because no other event with smaller time stamp can be produced at a port in $G(i)$. Note that for each

$i \in I$, given $\mathbb{C}_i$ and $G(i)$, the value of $\max_{p\in\mathbb{C}_i, o\in G(i)}\{d(p) - \delta(p,o) + \delta_0(i,o)\}$ can be computed statically.

We next sketch the proof that the strategy presented above cannot result in a deadlock, i.e., if a platform event queue is non-empty then after a finite amount of time an event from the queue will be safe to process. Assume there are $n \in \mathbb{N}$ events e_k in the queue with time stamps τ_k and waiting at input ports i_k to be processed to generate events at output ports o_k for $k = 1, ..., n$. The functions d, δ and δ_0 are bounded, so the condition 1) eventually becomes satisfied for each event in the queue. Assume that condition 1) is satisifed for all events in the queue, but no event satisfies condition 2). Let $\tau' = \min_{k=1,...,n}\{\tau_k + \delta(i_k, o_k)\}$ and let E' be the set of events e_k for which this minimum is achieved, i.e., let $E' = \{e_k \mid \tau_k + \delta(i_k, o_k) = \tau' \ , \ k = 1, ..., n\}$. This set contains at least two elements, because otherwise its only element would be safe to process being the unique event that could result in an event with time stamp τ'. Moreover, each element e_{k_1} from E' is declared unsafe because of another event e_{k_2} from E' such that $\delta(o_{k_2}, o_{k_1}) = 0$. Since this is true for every element in E', there exist $n' \leq n$ indices k_j such that $\delta(o_{k_{j+1}}, o_{k_j}) = 0$ for $j = 1, ..., n'$ and $k_{n'+1} = k_1$. Consequently, there exists a zero delay loop formed by ports $o_{k_1}, o_{k_2}, ..., o_{k_{n'}}$. This contradicts assumptions of our programming model introduced in Sec. II-A.

Note that strategy A does not assume that events at actor input ports are received in increasing time-stamp order. In particular, the network over which platforms communicate can reorder messages (events), so that events at a network output port are not received in the time-stamp order. However, if we assume there is no network reordering, then for each port of the system the events are received in the time-stamp order because actors themselves produce events in such an order. If this is the case, the execution strategy could be made simpler by considering for each path from the cut $\mathbb{C}_i$ only the event closest to $G(i)$. In particular, for a given path $\rho_{p,o}$ from $p \in \mathbb{C}_i$ to $o \in G(i)$ define: $\delta'(\rho_{p,o}) = \min\{\delta(p', o) \mid p' \in I \cap \rho_{p,o},$ $\text{input queue of p}' \text{ is non}-\text{empty}\}$, and let $p'(\rho_{p,o})$ be port p' for which this minimum is achieved. If there exists no such p' let $\delta'(\rho_{p,o}) = \infty$. The strategy for the case with additional assumption on networking is:

Strategy B. *An event at input port $i \in I$ with time stamp τ is safe to process if for each path $\rho_{p,o}$ from $p \in \mathbb{C}_i$ to $o \in G(i)$:*
the smallest time stamp in the queue of $p'(\rho_{p,o})$ is greater than

$$\tau - \delta'(\rho_{p,o}) + \delta_0(i,o) \quad for \quad \delta'(\rho_{p,o}) \neq \infty,$$

or physical time is greater than

$$\tau + d(p) - \delta(p,o) + \delta_0(i,o) \quad for \quad \delta'(\rho_{p,o}) = \infty.$$

Finally, we present another simplification to the strategy we described above. Here we require an event queue to be available for each platform to store and sort all events in the platform. Specifically, all events are sorted by the value of $\tau + \max_{o\in G(i)}(\delta_0(i,o))$, where τ is the time stamp of the event and i its input port. Intuitively, the event queue is sorted by the model time in which events are produced. Now, in contrast to earlier strategies, which consider multiple events for processing, in this implementation, only the first event in the queue is checked for safety. This greatly simplifies the execution strategy. In particular, if only the first event on the queue is considered for processing, there is no need to check the condition 2) of the general execution strategy from Sec. II-B because it will always be satisfied. However, since other events in the queue may be safe to process even when the first event is not, this approach may result in more conservative strategy. This depends both on the actor model and the characteristics of the input event sequences.

The safe-to-process analysis in this case can be simplified to a time stamp checking against physical time as follows:

Strategy C. *An event at input port $i \in I$ with time stamp τ is safe to process when the physical time has exceeded*

$$\tau + \max_{p\in\mathbb{C}_i, o\in G(i)}(d(p) - \delta(p,o) + \delta_0(i,o)).$$

If an event can be processed immediately, it is passed to the corresponding actor for processing. If no event can be processed immediately, the physical time at which an event can be processed can be determined, and a timed interrupt can be set to occur at that time.

Note that our execution strategy bears some resemblance to the one presented in [15]. However, the types of synchronization needed across platforms in both cases is largely different. Other than the transmission of data between platforms, [15] also requires synchronization barriers, which stall execution until all platforms arrive at such barriers, as well as cross-platform communication to test events for safe-to-process. Compared to periodic network synchronization packets that are required in our approach, we believe more network traffic will be a result of the execution strategy in [15]. Also, [15] assumes an upper bound of the difference between model time on different platforms, while we do not make such an assumption.

IV. PTIDES Strategies Comparison

The goal of this section is to compare the PTIDES strategies B and C discussed in the last section through a manufacturing assembly line example. The main difference between these strategies is that strategy B considers all events for processing, and assumes events always arrive in time stamp order, while strategy C only considers the event of smallest time stamp for processing, and does not assume events arrive in time stamp order.

Fig. 5 shows an example of a model that could be scheduled using the above strategies. This example is motivated by robotic arm control in manufacturing assembly lines, where robotic arms are programmed to perform certain functionalities in product construction. In this particular example, two robotic arms are controlled by *Platform2* and *Platform3*, respectively, and perform on a single product. Sensors reside in *Platform1*, and provide triggers needed for the arms. In this case, *SensorA* in *Platform1* generates events to tell both robotic arms that a

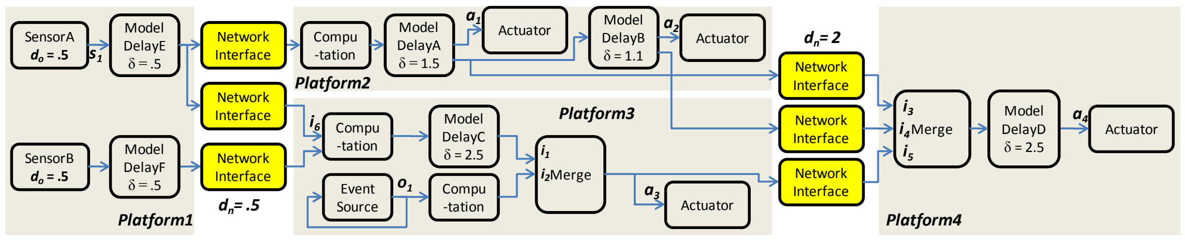

Fig. 5. Actor graph for assembly line example.

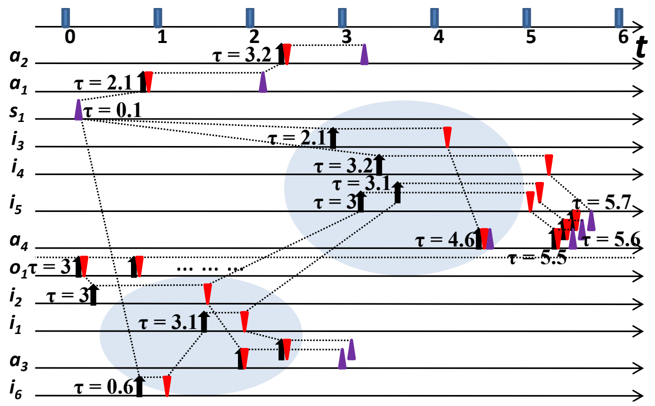

Fig. 6. Event trace of a simple PTIDES implementation.

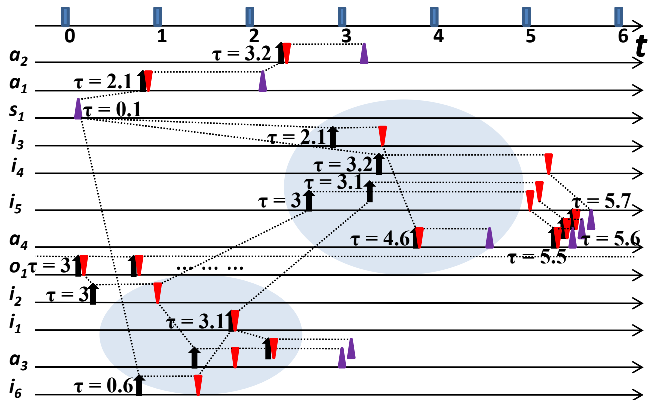

Fig. 7. Event trace of a more sophisticated PTIDES implementation.

half-completed product has arrived within reach, and sends these events across the first set of network interfaces. The arm controlled by *Platform2* then computes the exact position of the product, and performs required actuation when physical time is exactly equal to model time $\tau+.5+1.5$, where τ is the time stamp of the event sent by the sensor, and .5 and 1.5 are the model time delay introduced by *ModelDelayE* and *ModelDelayA* actors, respectively. The same event of time stamp τ is sent to *Platform3*, where *ModelDelayE* and *ModelDelayC* actors increases the time stamp by .5 and 2.5. Then the event is sent to a_3. The actuator subsequently performs the actuation at physical time $\tau+.5+2.5$. Back at *Platform2*, another actuation happens at physical time $\tau + .5 + 1.5 + 1.1$. While the three actuators at *Platform2* or *Platform3* await for physical time to equal model time to perform actuations, events are also sent across the set of network interfaces on the right, and to ports i_3, i_4 and i_5 at *Platform4*. These events in turn trigger some actuation at time stamps $\tau+.5+1.5+2.5$, $\tau+.5+1.5+1.1+2.5$, and $\tau+.5+2.5+2.5$, respectively. This actuation may trigger a shut down signal in case either of the robotic arms has failed to perform their functionalities correctly. Also notice that in *Platform3*, a separate path exists starting from the *Event Source* actor, which sends out periodic events through o_1. The periodic events may implement utilities the arm needs to perform periodically. Finally, *SensorB* in *Platform1* also sends events to *Platform3*, which can model a simple reset event that resets the position of the robotic arm.

Given this example, we present two event traces to show differences in execution strategies discussed in the previous subsections (Fig. 6 and 7). In the figures, the darken ellipses show differences for different strategies. Here, black arrows pointing up indicate an event becoming available (an event inserted into the corresponding input port), and red triangle pointing down indicate the event starts processing. Purple triangles pointing up are specific to real-time ports, where they either indicate an event is produced at a real-time output port, or an event is consumed for actuation at a real-time input port. In this scenario, we assume that each platform only has one processor to process events, thus resource contention exists. Also, here we assume actuation happens at the exact physical time that is equal to the time stamps of its input events, and they do not block the processor while waiting for the physical time. Finally, we assume it takes zero physical time for the scheduler to determine whether an event is safe to process.

The event trace in Fig. 7 corresponds to the strategy given in B, while the one in Fig. 6 corresponds to the strategy C. The ellipse on the left indicates a difference in event trace due to the fact that the simple strategy C only considers the event of smallest time stamp in the queue for processing, while the strategy B considers all events for processing.

For the case shown in the ellipse on the left, port i_6 first receives an event of time stamp 0.6, which according to strategy C, is safe to process at physical time $0.6 + .5 = 1.1$, where .5 is the d_o of the network interface. At the same time, an event of time stamp 3 is present at port i_2, which is safe-to-process at physical time $3 + .5 - 2.5 = 1$. In Fig. 6 the event at i_6 is first processed at physical time 1.1 because it generates the output of smaller time stamp. However in Fig. 7 the event at port i_2 is first processed at physical time 1, since it has a smaller safe-to-process time. This results in different processing times for event arriving at a_3. Thus, if model time delay of the *ModelDelayC* actor is descreased from 2.5 to 1.7, then for this event trace, actuator at *Platform3* would miss its deadline if scheduled by the strategy C, but not for the more sophisticated B. Also notice this also results in a large difference between when events are received at *Platform4*.

For the case shown in the ellipses on the right, i_3 first receives an event of time stamp 2.1, which is safe to process at physical time $2.1 + 2 = 4.1$, where 2 is the d_o of the second set of network interface. Since the strategy C does not assume events arrive in time stamp order across platforms, even though events are present at i_3, i_4, i_5, we still have to wait until physical time 4.1 to process the event at i_3, as shown in Fig. 6. Note, however, in Fig. 7, the strategy makes the additional assumption that events arrive in time stamp order across platforms, thus as soon as there is an event at each port of i_3, i_4, i_5, and the event at i_3 has the smallest time stamp, that event is processed. Again as in the last case, if the *ModelDelayD* actor is changed from 2.5 to 2, strategy B would still be able to meet the deadline for the event coming from i_3, while strategy C would not.

These traces show the strategy B is better compared to strategy C, due to its more sophisticated scheme to test whether an event is safe to process. However, note that we assumed zero computation time for the safe-to-process analysis in both cases, but the strategy C is much simpler than strategy B. Thus depending on the amount of computational power available and the input event trace, different conditions may require different strategies as to which one would make better use of the computational resource.

V. Feasibility Analysis

In this section we present sufficient conditions for a PTIDES implementation that applies strategy C to meet the real-time constraints defined in Sec. III-A. Since in this paper we study event safe-to-process analysis but do not focus on further event scheduling, in this section we assume that events are processed in zero time, i.e., actor execution takes zero time. We also assume that communication latencies and time synchronization errors between platforms are bounded and known. Meeting real-time constraints under these assumptions is what we here call feasibility property.

Let a discrete-event model $M = (P, \delta, d)$ be given with a set of ports P, model-time delay function δ (as defined in Sec. II-A) and physical-time delay function d (as defined in Sec. III-D). Note that the underlying graph of actors is given with the function δ. As explained in Sec. III-A, sensor and network latencies define the function d.

A *trace* of a model is a set of all events $e = (p, \tau, t)$ generated during an execution of the model, where p is event port, τ is timestamp of the event and t is physical time at which the event becomes available at the port p (is inserted in its queue).

Definition. Discrete-event model M is *feasible* if for each trace π of M such that each event of π at real-time output port satisfies condition III-A:(2), each event of π at real-time input port satisfies condition III-A:(1).

Before we discuss model feasibility in general case, we illustrate the sufficient condition for the simple model shown in Fig. 8a). Assume that the sensor maximum physical time delay d_o is smaller than the model-time delay $\delta(i_1, i_2)$. Let event $e_2 = (i_2, \tau_2, t_2)$ be an event at port i_2 generated by processing the event $e_1 = (i_1, \tau_1, t_1)$ at port i_1. From strategy C and the real-time output port assumption III-A:(2) we have $t_2 = \max\{t_1, \tau_1 + d_o - \delta(i_1, i_2) + \delta_0(i_1, i_2)\} = \max\{t_1, \tau_1 + d_o\} = \tau_1 + d_o$. Moreover, using the assumption $d_o \leq \delta(i_1, i_2)$ and the properties of the function δ we have $\tau_1 + d_o \leq \tau_1 + \delta(i_1, i_2) \leq \tau_2$. Therefore, if $d_o \leq \delta(i_1, i_2)$ then $t_2 \leq \tau_2$, i.e., the real-time input port constraint III-A:(1) is satisfied. According to the definition given above the model is feasible.

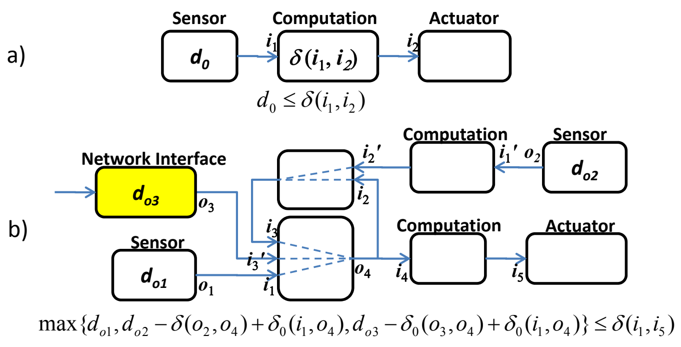

Fig. 8. Feasibility analysis illustration. Conditions are given for i_1.

The feasibility conditions that we propose next can independently be checked for each platform. Thus, to simplify the notation we first define the local model-time delay function $\underline{\delta} : P \times P \rightarrow \mathbb{R}^+ \cup \{\infty\}$. For two ports $p_1, p_2 \in P$ we have $\underline{\delta}(p_1, p_2) = \delta(p_1, p_2)$ if p_1 and p_2 belong to the same platform, or $\underline{\delta}(p_1, p_2) = \infty$ otherwise. In addition, let $\underline{O}$ and $\underline{I}$ denote the sets of real-time output and real-time input ports respectively. We call a path from a real-time output port to a real-time input port a *real-time path*.

Proposition. *Model $M = (P, \delta, d)$ is feasible if for each input port i on a real-time path*

$$\max_{\substack{o \in G(i) \\ \underline{o} \in \underline{O}}} \{d(\underline{o}) - \underline{\delta}(\underline{o}, o) + \delta_0(i, o)\} \leq \min_{\underline{i} \in \underline{I}} \{\underline{\delta}(i, \underline{i})\}.$$

The inequalities in Fig. 8 show feasibility conditions for port i_1 in two different models.

Proof. Let π be a trace of the model M. Assume first that all events in π are generated by processing at most one event (from π). Let $\underline{i} \in \underline{I}$ be an arbitrary real-time input port of the model and let $\underline{\pi} = (e_1, e_2, ..., e_n)$ be a sequence of events $e_k = (i_k, \tau_k, t_k)$ at input ports i_k $(k = 1, ..., n)$ such that e_k is generated by processing e_{k-1} $(k = 2, ..., n)$, $i_n = \underline{i}$ and i_1 is connected to a real-time output port. Note that there are no assumptions about the topology of the model, i.e., the underlying model graph can contain cycles. For instance, for a model shown in Fig. 8b) a sequence of input ports in $\underline{\pi}$ can be $(i_1, i_2, i_3, i_2, i_3, i_4, i_5)$. To prove the proposition we prove that $t_n \leq \tau_n$.

Lemma. *For $1 < j \leq n$ it holds*

$$t_j = \max_{1<k\leq j} \{\tau_{k-1} + \max_{\substack{o \in G(i_{k-1}) \\ \underline{o} \in \underline{O}}} \{d(\underline{o}) - \underline{\delta}(\underline{o}, o) + \delta_0(i_{k-1}, o)\}$$

Lemma Proof. Note first that for $1 \leq j < n$ it follows from the strategy C discussed in Sec. III-D

$$t_{j+1} = \max\{t_j, \tau_j + \max_{\substack{o \in G(i_j) \\ \underline{o} \in \underline{O}}} \{d(\underline{o}) - \underline{\delta}(\underline{o}, o) + \delta_0(i_j, o)\}$$

We use induction to prove the lemma. By the properties of the real-time output port connected to $i_1 \in G(i_1)$ we have (III-A:(2))

$$\tau_1 + \max_{\substack{o \in G(i_1) \\ \underline{o} \in \underline{O}}} \{d(\underline{o}) - \underline{\delta}(\underline{o}, o) + \delta_0(i_1, o)\} = \tau_1 + d_o \geq t_1,$$

$$t_2 = \max\{t_1, \tau_1 + \max_{\substack{o \in G(i_1) \\ \underline{o} \in \underline{O}}} \{d(\underline{o}) - \underline{\delta}(\underline{o}, o) + \delta_0(i_1, o)\}\}$$

$$= \tau_1 + \max_{\substack{o \in G(i_1) \\ \underline{o} \in \underline{O}}} \{d(\underline{o}) - \underline{\delta}(\underline{o}, o) + \delta_0(i_1, o)\}.$$

For the inductive step, first assume

$$t_j = \max_{1 < k \leq j} \{\tau_{k-1} + \max_{\substack{o \in G(i_{k-1}) \\ \underline{o} \in \underline{O}}} \{d(\underline{o}) - \underline{\delta}(\underline{o}, o) + \delta_0(i_{k-1}, o)\}.$$

Therefore,

$$t_{j+1} = \max\{\max_{1 < k \leq j} \{\tau_{k-1} + \max_{\substack{o \in G(i_{k-1}) \\ \underline{o} \in \underline{O}}} \{d(\underline{o}) - \underline{\delta}(\underline{o}, o) + \delta_0(i_{k-1}, o)\}\},$$

$$\tau_j + \max_{\substack{o \in G(i_j) \\ \underline{o} \in \underline{O}}} \{d(\underline{o}) - \underline{\delta}(\underline{o}, o) + \delta_0(i_j, o)\}.$$

$$= \max_{1 < k \leq j+1} \{\tau_{k-1} + \max_{\substack{o \in G(i_{k-1}) \\ \underline{o} \in \underline{O}}} \{d(\underline{o}) - \underline{\delta}(\underline{o}, o) + \delta_0(i_{k-1}, o)\}. \quad \square$$

Using the Lemma we have

$$t_n = \max_{1 < k \leq n} \{\tau_{k-1} + \max_{\substack{o \in G(i_{k-1}) \\ \underline{o} \in \underline{O}}} \{d(\underline{o}) - \underline{\delta}(\underline{o}, o) + \delta_0(i_{k-1}, o)\},$$

$$= \tau_K + \max_{\substack{o \in G(i_K) \\ \underline{o} \in \underline{O}}} \{d(\underline{o}) - \underline{\delta}(\underline{o}, o) + \delta_0(i_K, o)\},$$

for a certain K $(1 \leq K < n)$. Thus, from the proposition assumption and the properties of the δ (i.e., $\underline{\delta}$) function we have (III-A:(1))

$$t_n \leq \tau_K + \min_{\underline{i} \in \underline{I}} \{\underline{\delta}(i_K, \underline{i})\} \leq \tau_K + \underline{\delta}(i_K, i_n) \leq \tau_n.$$

In general, some events of a trace may be produced by processing multiple events. In that case these events must be with the same time stamp and available at the ports of the same port group. The proof would be similar as the one presented above, except that it would consist of the analysis on a tree structure instead of on the sequence $\underline{\pi}$. $\square$

VI. Summary and Future Work

We first presented a general execution strategy that enables correct event processing for timed models with discrete-event semantics. In our model, the timing assumptions and requirements for the components of a distributed embedded system define relationships between model time and physical time at real-time ports. Motivated by open and precise clock synchronization protocols that are becoming available for distributed real-time systems, we next presented several instances of the general execution strategy that make use of these relationships, clock synchronization and static program analysis. The strategies allow independent events to be processed out of time stamp order without backtracking and null message mechanism.

We are developing a simulation environment for the PTIDES programming model as an experimental domain in Ptolemy II [5]. We are also working on two PTIDES implementations, one on a set of Linux-based Agilent demo boxes equipped with the IEEE 1588 protocol, and the other using real-time Java environments. Our future work also include real-time schedulability problems for event schedulers without the assumptions of the analysis in Sec. V.

References

[1] F. Baccelli, G. Cohen, G. J. Olsder, and J.-P. Quadrat. *Synchronization and Linearity: An Algebra for Discrete Event Systems*. Wiley, 1992.

[2] A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in BIP. In *SEFM*, pages 3–12, Pune, 2006.

[3] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, 1991.

[4] K. M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transaction on Software Engineering*, 5(5), 1979.

[5] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity—the Ptolemy approach. *Proceedings of the IEEE*, 91(2):127–144, 2003.

[6] T. H. Feng and E. A. Lee. Real-time distributed discrete-event execution with fault tolerance. In *Proceedings of RTAS*, April 2008.

[7] R. M. Fujimoto. Parallel discrete event simulation. *Commun. ACM*, 33(10):30–53, 1990.

[8] IEEE. 1588: IEEE standard for a precision clock synchronization protocol for networked measurement and control systems, 2002.

[9] A. Jantsch and I. Sander. Models of computation and languages for embedded system design. *IEEE Proceedings on Computers and Digital Techniques*, 152(2):114–129, 2005.

[10] D. Jefferson. Virtual time. *ACM Trans. Programming Languages and Systems*, 7(3):404–425, 1985.

[11] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty. Model-integrated development of embedded software. *Proceedings of the IEEE*, 91(1):145–164, 2003.

[12] E. A. Lee. Discrete event models: Getting the semantics right. In *Proceedings of WSC*. Winter Simulation Conference, 2006.

[13] E. A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[14] E. A Lee and H. Zheng. Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In *EMSOFT*, Salzburg, Austria, 2007. ACM.

[15] B. D. Lubachevsky. Bounded lag distributed discrete event simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation*, volume 19, San Diego, CA, 1988.

[16] R. Münzenberger, M. Drfel, R. Hofmann, and F. Slomka. A general time model for the specification and design of embedded real-time systems. *Microelectronics Journal*, 34:989–1000, 2003.

[17] K. Strehl, L. Thiele, M. Gries, D. Ziegenbein, R. Ernst, and J. Teich. Funstatean internal design representation for codesign. *IEEE Trans. on VLSI Systems*, 9(4):524–544, 2001.

[18] Y. Zhao, J. Liu, and E. A. Lee. A programming model for time-synchronized distributed real-time systems. In *Proceedings of RTAS*, pages 259–268, Bellevue, WA, USA, Apr 2007.

[19] Y. Zhou and E. A. Lee. Causality interfaces for actor networks. *ACM Transactions on Embedded Computing Systems (TECS)*, 2008.