Toward an Effective Execution Policy for Distributed Real-Time Embedded Systems

Thomas Huining Feng, Edward A. Lee, Hiren D. Patel, and Jia Zou
Center for Hybrid and Embedded Software Systems, EECS
University of California, Berkeley
Berkeley, CA 94720, USA
{tfeng,eal,hiren,jiazou}@eecs.berkeley.edu

***Abstract*—Zhao, Liu, and Lee have proposed using a discrete-event (DE) model of computation as a programming model for distributed real-time embedded systems. The advantage of using DE is that it provides a semantic foundation that is simple, time-aware, deterministic and natural as a specification language for many applications. This programming model is based on a carefully chosen relationship between DE's model time and real time (physical time). We define here a criterion that preserves conservative execution (thus not requiring backtracking) while allowing for concurrent and distributed execution. The classic Chandy and Misra technique is one execution policy that satisfies the criterion, but the criterion explicitly allows many other alternatives. We discuss alternatives that offer more concurrency than Chandy and Misra and that exploit time synchronization to eliminate the need for null messages.**

I. Introduction

Current programming practices for distributed real-time embedded systems often employ commercial-off-the-shelf real-time operating systems (RTOS) and real-time object request brokers as utilities for implementing the system. Programmers also use languages such as C with concurrency expressed by threads. RTOSs and threads however, provide only weak guarantees that the system will meet real-time constraints. They also do not guarantee that the behavior of the system is deterministic. A consequence is that the only way to achieve confidence in the implementation is through extensive testing. This validates that the functionality and real-time requirements of the system are met for the tested scenarios. However this technique is inherently flawed, because no assurance can be given about the behavior of the entire system. We identify the source of the problem for such techniques as the lack of a timed semantic foundation combined with the inherent nondeterminism in threads [1].

These problems can be addressed by using a distributed discrete-event (DE) model of computation (MoC) [2]. Though normally used for simulation (of hardware, networks, and systems of systems, for example), by carefully binding real time with model time at sensors, actuators, and network interfaces, DE can be used for distributed embedded systems [3]. The advantage of using DE as a semantic foundation is that it is simple, time-aware, deterministic, and natural as a specification language for many applications.

Distributed DE simulation is an old topic [2]. 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], [5]. 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 [6] 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) [7].

Established conservative techniques however, also prove inadequate. In the classic Chandy and Misra technique [4], [5], 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." These messages increase networking overhead and also reduce the available precision of real-time constraints. Moreover, the technique is not robust; failure of single component results in no more such messages, thus blocking progress in other components. Our work is related to several efforts to reduce the number of null messages, such as [8], but makes much heavier use of static analysis.

The key idea of Zhao, Liu and Lee in [3] is to leverage static analysis of DE models to achieve distributed DE scheduling

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 and AF-TRUST #FA9550-06-1-0244), the Air Force Research Lab (AFRL), the State of California Micro Program, and the following companies: Agilent, Bosch, DGIST, National Instruments, and Toyota.

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. In this case, the mere passage of time obviates the need for null messages.

By extending the work of [3] we are moving toward defining a programming model that 1) builds on top of a strong timed semantic foundation, 2) maximizes concurrency of the implementation, 3) provides deterministic schedulability analysis, and 4) eases specification of real-time constraints. We call the programming model PTIDES (pronounced "tides," where the "P" is silent, as in "Ptolemy"), an acronym for *programming temporally integrated distributed embedded systems*. In this work-in-progress paper however, we only elaborate on the carefully chosen relationship between model time and real time, and then present our formulation of a general execution strategy for a PTIDES specification.

II. Model Time and Physical Time

In our DE MoC, actors are concurrent components with input and output ports. The input ports receive time-stamped messages from other actors, and the output ports send time-stamped messages to other actors. Actors react to input messages by "firing," by which we mean performing a finite computation and possibly sending output messages. An actor may also send a time-stamped message to itself, effectively requesting a future firing.

The "time" in time stamps is *model time*, not *physical time*. DE semantics is agnostic about when in physical time time-stamped events are processed. All that matters is that each actor process input events in time-stamp order. That is, if it fires in response to an input event with time stamp t, it should not later fire in response to an input event with time stamp less than t.

The semantics of DE models is studied in [9], [10], [11], [12]. In particular, the structure of model time is important for dealing correctly with simultaneous events and feedback systems. For the purposes of this paper, we only care that there are policies for dealing predictably with multiple events with identical time stamps. To be concrete, we will assume that time stamps are elements of the set $\mathbb{R}^+ \cup \{\infty\}$. In full generality, however, our techniques work for any set of time stamps that is totally ordered, has a top and a bottom, and has a closed addition operator.

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. To distinguish it from model time, we refer to it as *physical time* or *real time*. Here, we assume a classical Newtonian notion of physical time, and assume that each compute platform in a distributed system maintains a clock that measures the passage of physical time. These clocks are not perfect, so each platform has a distinct local notion of physical time. We assume further that we can find a bound on the discrepancies between clocks on different platforms. That is, at any global instant, any two clocks in the system agree on the notion of physical time up to some bounded error.

Synchronized clocks turn out to be quite practical [13]. We have had available for some time generic clock synchronization protocols like NTP [14]. Recently, however, techniques have been developed that deliver astonishing precision, such as IEEE 1588 [15]. Hardware interfaces for Ethernet have recently become available that advertise a precision of 8ns over a local area network. Such precise clock synchronization offers truly game-changing opportunities for distributed embedded software.

We assume that model time and physical time are disjoint, but that they can be compared. That is, we assume that model time is in fact a representation of physical time, even though time-stamped events can occur at arbitrary physical times. In our DE models, an actor that wraps a sensor, however, cannot produce time-stamped events at arbitrary times. In particular, it will produce a time-stamped output only after physical time (the local notion of physical time) equals or exceeds the value of the time stamp. That is, the time stamp represents the physical time at which the sensor reading is taken, and hence it cannot appear at a physical time earlier than the value of the time stamp.

An actor that wraps an actuator has a complementary constraint. A time-stamped input to such an actor will be interpreted as a command to produce a physical effect at (local) physical time equal to the time stamp. Consequently, the model-time time stamp is a physical-time deadline for delivery of an event to an actuator.

At actors that are neither sensors or actuators, there is no relationship between physical and model time. At these actors, input events must be processed in model-time order, but such processing can occur at any physical time (earlier or later than the time stamp).

III. The PTIDES Execution Strategy

Following [3], we capture the information of minimum model-time delay with *relevant dependency* [3]. In our formal representation of actor-oriented models, a model consists of a set A of actors. Any actor $\alpha \in A$ has a set of input ports I_α and a set of output ports O_α. Without loss of generality, we assume I_α and O_α to be disjoint. We also assume that any local state maintained by the actor appears at an output port, so we do not need to address it explicitly. We further assume that ports are interconnected by a fixed, static network, where each input port is connected to at most one output port. This will ensure that all data dependencies are relations between ports. The set of all input ports is $I = \bigcup_{\alpha \in A} I_\alpha$, the set of all output ports is $O = \bigcup_{\alpha \in A} O_\alpha$, and the set of all ports is $P = I \cup O$.

The *minimum delay* (in model time) is defined as function $\delta : P \times P \rightarrow \mathbb{R}^+ \cup \{\infty\}$, where $\mathbb{R}^+$ is the set of non-negative real numbers. For $p_1, p_2 \in P$, $\delta(p_1, p_2)$ is the minimum

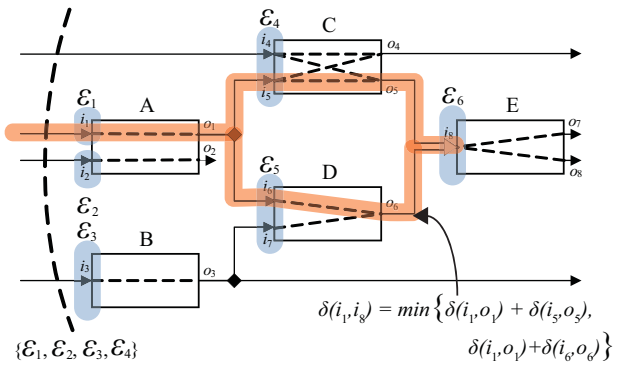

Fig. 1. Example with Minimum Delay, Relevant Dependency and Cuts

difference between the model time stamp of any event e_1 at p_1 and that of any event e_2 at p_2 that totally or partially depends on e_1. Intuitively, this number represents the delay (in model time, not physical time) that it takes for e_1 at p_1 to influence any event at p_2. If no event at p_2 depends on the events at p_1, then we define $\delta(p_1, p_2) = \infty$.

We assume that for every actor α, $\delta(p_i, p_j)$ is known for all $p_i \in I_\alpha$ and $p_j \in O_\alpha$. This information constitutes an interface definition for the actor [16]. To compose these interfaces, we use a min-plus algebra [17] to compute δ for any pair of ports based on [3]. The min-plus algebra aggregates these dependencies over multiple paths between ports.

We define a *path* from port p_1 to p_n to be a sequence of ports $[p_1, p_2, \cdots, p_n]$, where for any j $(1 \leq j < n)$, either p_j is directly connected to p_{j+1}, or $p_j \in I_\alpha$ and $p_{j+1} \in O_\alpha$ for some actor α and $\delta(p_j, p_{j+1}) < \infty$. A *subpath* is a sequence of consecutive ports in a path. For any pair of ports p_1, p_n, the minimum delay $\delta(p_1, p_n)$ is the minimum of the total delays on all the paths from p_1 to p_n.

An example of calculating the minimum delay is provided in Figure 1. The input ports are labeled i_1 through i_8 and the output ports are labeled o_1 through o_8. The actors are represented by rectangles. A triangle pointing into an actor denotes an input port. (i_8 is a *multi-port* denoted by a hollow triangle, which accepts multiple input connections. It can be represented as multiple separate ports in our formulation.) The minimum delay between i_1 and i_8, $\delta(i_1, i_8)$, can be computed by $\min\{\delta(i_1, o_1) + \delta(i_5, o_5), \delta(i_1, o_1) + \delta(i_6, o_6)\}$. (Direct connections, such as the one between o_1 and i_5, do not incur any delay in model time.)

An actor may have an output port at which events never depend on events at some of its input ports. This leads us to partition the set of input ports I into equivalence classes $\mathbb{E} = \{\mathcal{E}_1, \mathcal{E}_2, \cdots, \mathcal{E}_k\} \subseteq 2^I$. We first define relation $\sim$ such that for any two ports i_1 and i_2, $i_1 \sim i_2$ if and only if they are both in I_α for some actor α and there exists an output port $o \in O_\alpha$ such that $\delta(i_1, o) < \infty$ and $\delta(i_2, o) < \infty$. An *equivalence class* is then a transitive closure of the $\sim$ relation. Intuitively, if i_1 and i_2 are in an equivalence class, then 1) they belong to the same actor, and 2) the events received at them directly or indirectly influence the output signal of an output port of that actor. This means that these events must be processed in time stamp order. If i_1 and i_2 are not in the same equivalence class, then the input events at i_1 can be processed independently of those at i_2, and vice versa.

We now define *relevant dependency* [3] to be function $d : \mathbb{E} \times \mathbb{E} \to \mathbb{R}^+ \cup \{\infty\}$. For $\mathcal{E}_j, \mathcal{E}_k \in \mathbb{E}$,

$$d(\mathcal{E}_j, \mathcal{E}_k) = \min_{i_m \in \mathcal{E}_j, i_n \in \mathcal{E}_k} \{\delta(i_m, i_n)\}$$

As an example, in Figure 1, $\mathcal{E}_1$ through $\mathcal{E}_6$ are equivalence classes. The relevant dependency between $\mathcal{E}_4$ and $\mathcal{E}_6$ is $\min\{\delta(i_4, i_8), \delta(i_5, i_8)\} = \min\{\delta(i_4, o_5), \delta(i_5, o_5)\}$.

The relevant dependency function is pre-computed in a static analysis before execution. Based on this information, we can execute a DE model according to the PTIDES execution strategy, which we discuss in this section.

A set $\mathbb{C}_\mathcal{E} \subseteq \mathbb{E}$ is called a *dependency cut* for equivalence class $\mathcal{E}$ [7] if it is a minimal set of equivalence classes that satisfies the following condition.

> *For any input port $i \in \mathcal{E}$ and any path ρ to i, there exist $\mathcal{E}' \in \mathbb{C}_\mathcal{E}$, input port $i' \in \mathcal{E}'$ and a path ρ' from i' to i, such that either ρ is a subpath of ρ' or ρ' is a subpath of ρ.*

Intuitively, a dependency cut for $\mathcal{E}$ is a "complete" set of equivalence classes on which $\mathcal{E}$ depends. Completeness in this case means that for each port in $\mathcal{E}$, all ports it depends on will be accounted for in $\mathbb{C}_\mathcal{E}$, either directly by being included or indirectly by having either upstream or downstream ports included. Again using Figure 1 as an example, the dashed curve depicts one possible dependency cut for $\mathcal{E}_6$, namely $\mathbb{C}_{\mathcal{E}_6} = \{\mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3, \mathcal{E}_4\}$. Note that an equivalence class $\mathcal{E}$ can have many distinct dependency cuts. The dependency cut is not unique. Note further that $\{\mathcal{E}\}$ is always a (trivial) dependency cut for $\mathcal{E}$.

A dependency cut can be used to determine when an actor can fire. Specifically, given a dependency cut $\mathbb{C}_\mathcal{E}$, the actor α to which the ports in $\mathcal{E}$ belong determines whether it can process input events received at the ports in $\mathcal{E}$ with model time stamps less than or equal to t using the following strategy [7]:

> *If for any $\mathcal{E}' \in \mathbb{C}_\mathcal{E}$, α has received all events at the ports in $\mathcal{E}$ that depend on events at the ports in $\mathcal{E}'$ with model times smaller than $t - d(\mathcal{E}', \mathcal{E})$, then it can fire and process the input event received at a port in $\mathcal{E}$ with smallest model time (among all the available events at the ports in $\mathcal{E}$) that is less than or equal to t.*

This principle, of course, can be satisfied by a classical DE scheduler, which uses a global event queue to sort events by time stamp. In this case, the oldest event (with the least time stamp) can always be processed[1]. However, this principle relaxes the policy considerably, clarifying that we only need to know whether an event is "oldest" among the events that

[1]This assumes, of course, that all actors are causal, so events that are produced in reaction to processing an event always have a time stamp at least as great as that of the processed event.

can appear in a dependency cut. We do not need to know that it is globally oldest.

The classic distributed DE execution strategy of Chandy and Misra [4], [5] uses multiple event queues, one on each execution platform. The technique is equivalent to defining the dependency cut to include the ports at the boundaries between platforms. It then simply assumes that all events with time stamps up to that of the most recently received event have been seen. This technique requires messages to be received in order to make progress, hence the requirement for null messages.

The technique of Zhao, Liu, and Lee [3] augments the Chandy and Misra model with an assumption that real-time clocks on the distributed platforms are synchronized up to some bounded error. It further imposes relationships between real time and model time at sensors and actuators. It then uses relevant dependency analysis to determine at any given real time that all events at the boundary ports have been seen with time stamps up real time minus a statically calculated offset.

An obvious extension would combine these two techniques. Non-real-time portions of a DE model may use a technique like Chandy and Misra while real-time portions use a technique like Zhao, Liu, and Lee. The above principle allows for freely intermixing these. If the non-real-time portions can be shown to be sufficiently "ahead of time," then the use of Chandy and Misra would not compromise the ability to meet real-time constraints.

More interestingly, the above principle allows for other choices of dependency cuts. Putting a dependency cut on the boundary between platforms imposes a constraint that either events traversing that boundary have real-time constraints or that null messages are used. The above principle, however, allows choices other than at the boundaries.

Another possibility is to offer system designers explicit control over the relationship between model time and real time at the platform boundaries. For example, a *NetworkInterface* actor might be defined to have input ports like those of an actuator, which impose a real-time constraint on events delivered to those ports. Specifically, we require that events delivered to the network interface with time stamp t be delivered at physical time less than or equal to t. If we further assume a bounded network delay N_{delay} for a message to be sent across the network, then the receiving platform is guaranteed to receive those events at real time no later than $t+N_{delay}$. This real time is in terms of the sending platform's local clock, but using a time synchronization protocol with bounded error, such as IEEE 1588 [15], the receiving platform can decide a lower bound of the time stamps of future input events by merely checking its own local clock. This allows it to independently determine whether it can process events that it has already received. If all network communication links use network interfaces, then scheduling and schedulability analysis becomes separable by platform.

Another possible objective could be to choose dependency cuts to facilitate schedulability analysis. In particular, whether we have worst-case execution time information or not for particular actors could affect the choice of dependency cut, and hence affect how the distributed model is executed.

IV. Conclusion

We have defined a correctness principle for conservative execution of a distributed discrete-event model that is suitable for both classical distributed simulation and for distributed real-time execution. Our correctness principle relies on a choice of *dependency cut*. The principle can be applied in a variety of ways, obtaining previously given techniques as special cases, but also clarifying that there are many more alternatives. A remaining challenge is to formulate appropriate optimization problems that guide the application of the principle, to solve these optimization problems, and to provide a distributed execution engine that implements them.

References

[1] E. A. Lee, "The problem with threads," *Computer*, vol. 39, no. 5, pp. 33–42, 2006.

[2] R. M. Fujimoto, "Parallel discrete event simulation," *Communications of the ACM*, vol. 33, no. 10, pp. 30–53, 1990.

[3] Y. Zhao, J. Liu, and E. A. Lee, "A programming model for time-synchronized distributed real-time systems," in *Proceedings of the 13th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 07)*, Bellevue, WA, USA, April 2007, pp. 259–268.

[4] K. M. Chandy and J. Misra, "Distributed simulation: A case study in design and verification of distributed programs," *IEEE Transaction on Software Engineering*, vol. 5, no. 5, 1979.

[5] J. Misra, "Distributed discrete-event simulation," *ACM Computing Surveys*, vol. 18, no. 1, pp. 39–65, 1986.

[6] D. Jefferson, "Virtual time," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 3, pp. 404–425, 1985.

[7] T. H. Feng and E. A. Lee, "Real-time distributed discrete-event execution with fault tolerance," in *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 08)*, St. Louis, MO, USA, April 2008.

[8] R. D. Vries, "Reducing null messages in Misra's distributed discrete event simulation method," *IEEE Transactions on Software Engineering*, vol. 16, no. 1, pp. 82–91, 1990.

[9] E. A. Lee, "Modeling concurrent real-time processes using discrete events," *Annals of Software Engineering*, vol. 7, pp. 25–45, 1999.

[10] E. A. Lee and H. Zheng, "Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems," in *Proceedings of the 7th ACM & IEEE International Conference on Embedded Software (EMSOFT 07)*. ACM, October 2007, pp. 114–123.

[11] X. Liu and E. A. Lee, "CPO semantics of timed interactive actor networks," UC Berkeley, Technical Report EECS-2006-67, May 18 2006.

[12] X. Liu, E. Matsikoudis, and E. A. Lee, "Modeling timed concurrent systems," in *CONCUR 2006 - Concurrency Theory*, vol. LNCS 4137. Bonn, Germany: Springer, August 27-30 2006.

[13] S. Johannessen, "Time synchronization in a local area network," *IEEE Control Systems Magazine*, pp. 61–69, 2004.

[14] D. L. Mills, "A brief history of NTP time: Confessions of an internet timekeeper," *ACM Computer Communications Review*, vol. 33, 2003.

[15] IEEE Instrumentation and Measurement Society, "1588: IEEE standard for a precision clock synchronization protocol for networked measurement and control systems," IEEE, Standard Specification, November 8 2002.

[16] L. de Alfaro and T. A. Henzinger, "Interface theories for component-based design," in *First International Workshop on Embedded Software (EMSOFT 01)*, vol. LNCS 2211. Lake Tahoe, CA: Springer-Verlag, October 2001, pp. 148–165.

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