Reinventing Computing for Real Time

Edward A. Lee and Yang Zhao

University of California, Berkeley
{eal,ellen_zh}@eecs.berkeley.edu

Abstract. This paper studies models of computation, software techniques, and analytical models for distributed timed systems. By "timed systems" we mean those where timeliness is an essential part of the behavior. By "distributed systems" we mean computational systems that are interconnected on a network. Applications of timed distributed systems include industrial automation, distributed immersive environments, advanced instrumentation systems, networked control systems, and many modern embedded software systems that integrate networking. The introduction of network time protocols such as NTP (at a coarse granularity) and IEEE 1588 (at a fine granularity) makes possible time coherence that has not traditionally been part of the computational models in networked systems. The main question we address in this paper is: Given time synchronization with some known precision, how does this change how distributed applications are designed and developed? A second question we address is: How can time synchronization help with realizing coordinated real-time events.

1 Introduction

Despite considerable progress in software and hardware techniques, when embedded computing systems absolutely must meet tight timing constraints, many of the advances in computing become part of the problem, not part of the solution. Although synchronous digital logic delivers precise timing determinacy, advances in computer architecture and software have made it difficult or impossible to estimate or predict the execution time of software. Moreover, networking techniques introduce variability and stochastic behavior, and operating systems rely on best effort techniques. Worse, programming languages lack time in their semantics, so timing requirements are only specified indirectly. This paper studies methods for programming ensembles of networked real-time, embedded computers where time and concurrency are first-class properties of the program.

This contrasts with established software techniques, where time and concurrency are afterthoughts. The prevailing view of real-time appears to have been established well before embedded computing was common. Wirth reduces real-time programming to threads with bounds on execution time, arguing that "it is prudent to extend the conceptual framework of sequential programming as little as possible and, in particular, to avoid the notion of execution time" [30]. In this sequential framework, "computation" is accomplished by a terminating sequence of state transformations. This core abstraction underlies the design of nearly all

F. Kordon and J. Sztipanovits (Eds.): Monterey Workshop 2006, LNCS 4322, pp. 1–25, 2007.

computers, programming languages, and operating systems in use today. But unfortunately, this core abstraction does not fit embedded software very well.

This core abstraction fits reasonably well if embedded software is simply "software on small computers." In this view, embedded software differs from other software only in its resource limitations (small memory, small data word sizes, and relatively slow clocks). In this view, the "embedded software problem" is an optimization problem. Solutions emphasize efficiency; engineers write software at a very low level (in assembly code or C), avoid operating systems with a rich suite of services, and use specialized computer architectures such as programmable DSPs and network processors that provide hardware support for common operations. These solutions have defined the practice of embedded software design and development for the last 25 years or so. In an analysis that remains as valid today as 18 years ago, Stankovic laments the resulting misconceptions that real-time computing "is equivalent to fast computing" or "is performance engineering" [29].

Of course, thanks to the semiconductor industry's ability to follow Moore's law, the resource limitations of 25 years ago should have almost entirely evaporated today. Why then has embedded software design and development changed so little? It may be that extreme competitive pressure in products based on embedded software, such as consumer electronics, rewards only the most efficient solutions. This argument is questionable, however. There are many examples where functionality has proven more important than efficiency. It is arguable that resource limitations are not the only defining factor for embedded software, and may not even be the principal factor.

Stankovic argues that "the time dimension must be elevated to a central principle of the system. Time requirements and properties cannot be an afterthought" [29]. But in mainstream computing, this has not happened. The "time dimension," of course, is inextricably linked to concurrency, and prevailing models of concurrency (threads and message passing) are in fact obstacles to elevating time to a central principle.

In embedded software, several recent innovations provide unconventional ways of programming concurrent and/or timed systems. We point to six cases that define concurrency models, component architectures, and management of time-critical operations in ways significantly different from prevailing software engineering techniques. The first is nesC with TinyOS [8], which was developed for programming very small programmable sensor nodes called "motes." The second is Click [16], which was created to support the design of software-based network routers. These first two have an imperative flavor, and components interact principally through procedure calls. The third is Simulink with Real-Time Workshop (from The MathWorks), which was created for embedded control software and is widely used in the automotive industry. The fourth is SCADE (from Esterel Technologies, see [2], which was created for safety-critical embedded software and is used in avionics. These two have a more declarative flavor, where components interact principally through messages rather than procedure calls. The fifth is the family of hardware description languages, including Verilog, VHDL, and

SystemC, which express vast amounts of concurrency, principally using discrete-event semantics. The sixth example is LabVIEW, from National Instruments, a dataflow programming environment with a visual syntax designed for embedded instrumentation applications. The amount and variety of experimentation with alternative models of computation for embedded systems is yet a further indication that the prevailing software abstractions are inadequate.

The approach in this paper leverages the concept of actor-oriented design [20], borrowing ideas from Simulink and from Giotto [12], an experimental real-time programming language. However, it addresses a number of limitations in Simulink and Giotto by building similar multitasking implementations from specifications that combine dataflow modeling and distributed discrete-event modeling. In discrete-event models, components interact with one another via events that are placed on a time line. Some level of agreement about time across distributed components is necessary for this model to have a coherent semantics. While distribution of discrete-event models has long been used to exploit parallel computing to accelerate execution [31], we are not concerned here with accelerating execution. The focus is instead on using a model of time as a binding coordination agent. This steers us away from conservative techniques (like Chandy and Misra [3]) and optimistic techniques (like Time Warp [15]). One interesting possibility is based on distributed consensus (as in Croquet [28]). In this paper, we focus on techniques based on distributing discrete-event models, with functionality specified by dataflow models. Our technique allows out of order execution without sacrificing determinacy and without requiring backtracking. The use of dataflow formalisms [26] supports mixing untimed and event-triggered computation with timed and periodic computation.

2 Embedded Software

There are clues that embedded software differs from other software in quite fundamental ways. If we examine carefully why engineers write embedded software in assembly code or C, we discover that efficiency is not the only concern, and may not even be the main concern. The reasons may include, for example, the need to count cycles in a critical inner loop, not to make it fast, but rather to make it predictable. No widely used programming language integrates a way to specify timing requirements or constraints. Instead, the abstractions they offer are about scalability (inheritance, dynamic binding, polymorphism, memory management), and if anything further obscure timing (consider the impact of garbage collection on timing). Counting cycles, of course, becomes extremely difficult on modern processor architectures, where memory hierarchy (caches), dynamic dispatch, and speculative execution make it nearly impossible to tell how long it will take to execute a particular piece of code. Embedded software designers may choose alternative processor architectures such as programmable DSPs not only for efficiency reasons, but also for predictability of timing.

Another reason engineers stick to low-level programming is that embedded software has to interact with hardware that is specialized to the application.

In conventional software, interaction with hardware is the domain of the operating system. Device drivers are not typically part of an application program, and are not typically created by application designers. But in the embedded software context, generic hardware interfaces are rarer. The fact is that creating interfaces to hardware is not something that higher level languages support. For example, although concurrency is not uncommon in modern programming languages (consider threads in Java), no widely used programming language includes in its semantics the notion of interrupts. Yet the concept is not difficult, and it can be built into programming languages (consider for example nesC [8] and TinyOS [13], which are widely used for programming sensor networks).

It becomes apparent that the avoidance of so many recent improvements in computation is not due to ignorance of those improvements. It is due to a mismatch of the core abstractions and the technologies built on those core abstractions. In embedded software, time matters. In the 20th century abstractions of computing, time is irrelevant. In embedded software, concurrency and interaction with hardware are intrinsic, since embedded software engages the physical world in non-trivial ways (more than keyboards and screens). The most influential 20th century computing abstractions speak only weakly about concurrency, if at all. Even the core 20th century notion of "computable" excludes all interesting embedded software, since to be "computable" you must terminate. In embedded software, termination is failure.

Embedded systems are integrations of software and hardware where the software reacts to sensor data and issues commands to actuators. The physical system is an integral part of the design and the software must be conceptualized to operate in concert with that physical system. Physical systems are intrinsically concurrent and temporal. Actions and reactions happen simultaneously and over time, and the metric properties of time are an essential part of the behavior of the system. Prevailing software methods abstract away time, replacing it with ordering. In imperative languages such as C, C++, and Java, the order of actions is defined by the program, but not their timing. This prevailing imperative abstraction is overlaid with another, that of threads or processes, typically provided by the operating system, but occasionally by the language (as in Java).

The lack of timing in the core abstraction is a flaw, from the perspective of embedded software, and threads as a concurrency model are a poor match for embedded systems. They are mainly focused on providing an illusion of parallelism in fundamentally sequential models, and they work well only for modest levels of concurrency or for highly decoupled systems that are sharing resources, where best-effort scheduling policies are sufficient. Indeed, none of the six examples given above include threads or processes in the programmer's model.

Embedded software systems are generally held to a much higher reliability standard than general purpose software. Often, failures in the software can be life threatening (e.g., in avionics and military systems). The prevailing concurrency model in general-purpose software that is based on threads does not achieve adequate reliability [19]. In this prevailing model, interaction between threads is extremely difficult for humans to understand. The basic techniques for controlling

this interaction use semaphores and mutual exclusion locks, methods that date back to the 1960s [5] and 1970s [14]. These techniques often lead to deadlock or livelock. In general-purpose computing, this is inconvenient, and typically forces a restart of the program (or even a reboot of the operating system). However, in embedded software, such errors can be far more than inconvenient. Moreover, software is often written without sufficient use of these interlock mechanisms, resulting in race conditions that yield nondeterministic program behavior. In practice, errors due to misuse (or no use) of semaphores and mutual exclusion locks are extremely difficult to detect by testing. Code can be exercised for years before a design flaw appears. Static analysis techniques can help (e.g. Sun Microsystems' LockLint), but these methods are often thwarted by conservative approximations and/or false positives, and they are not widely used in practice.

It can be argued that the unreliability of multithreaded programs is due at least in part to inadequate software engineering processes. For example, better code reviews, better specifications, better compliance testing, and better planning of the development process can help solve the problems. It is certainly true that these techniques can help. However, programs that use threads can be extremely difficult for programmers to understand. If a program is incomprehensible, then no amount of process improvement will make it reliable. Formal methods can help detect flaws in threaded programs, and in the process can improve the understanding that a designer has of the behavior of a complex program. But if the basic mechanisms fundamentally lead to programs that are difficult to understand, then these improvements will fall short of delivering reliable software. Incomprehensible software will always be unreliable software.

Prevailing practice in embedded software relies on bench testing for concurrency and timing properties. This has worked reasonably well, because programs are small, and because software gets encased in a box with no outside connectivity that can alter the behavior. However, applications today demand that embedded systems be feature-rich and networked, so bench testing and encasing become inadequate. In a networked environment, it becomes impossible to test the software under all possible conditions. Moreover, general-purpose networking techniques themselves make program behavior much more unpredictable.

What would it take to achieve concurrent and networked embedded software that was absolutely positively on time, to the resolution and reliability of digital logic? Unfortunately, everything would have to change. The core abstractions of computing need to be modified to embrace time. Computer architectures need change to deliver precisely timed behaviors. Networking techniques need to change to provide time concurrence. Programming languages have to change to embrace time and concurrency in their core semantics. Operating systems have to change to rely less on priorities to (indirectly) specify timing requirements. The separation of operating systems from languages has to be rethought. Software engineering methods need to change to specify and analyze the temporal dynamics of software. What is needed is nearly a reinvention of computer science.

No individual project, obviously, could possibly take all of this on. Fortunately, there is quite a bit of prior work to draw on. To name a few examples, architecture

techniques such as software-managed caches promise to deliver much of the benefit of memory hierarchy without the timing unpredictability [1,6]. Operating systems such as TinyOS [13] provide simple ways to create thin wrappers around hardware. Programming languages such as Lustre/SCADE [2,11] provide understandable and analyzable concurrency. Embedded software languages such as Simulink provide time in their semantics. Our own prior work shows how to generate hard-real time code from dataflow graphs [27].

In this paper, we focus on programming languages, pursuing abstractions that include time and concurrency as first-class properties, creating mechanisms for programming ensembles of networked embedded computers, rather than just programming individual computers, and creating mechanisms for tightly integrating hardware behavior into programs. We focus on applications in instrumentation and in distributed gaming; the first of these requires more precise timing synchronization, so we will leverage the new IEEE 1588 standard, which provides time synchronization across ethernet networks at resolutions down to tens of nanoseconds. The second requires time synchronization at more human scales, large fractions of a second, and may be able to effectively use time synchronization protocols such as NTP (network time protocol).

3 Concurrency and Time

We will focus on ways of giving programs where concurrency and time are essential aspects of a design, and most particularly on ways of compiling such programs to produce deployable real-time code. Time is a relatively simple issue, conceptually, although delivering temporal semantics in software can be challenging. Time is about the ordering of events. Event x happens before event y, for example. But in embedded software, time also has a metric. That is, there is an amount of time between events x and y, and the amount of time may be an important part of the correctness of a system.

In software, it is straightforward to talk about the order of events, although in concurrent systems it can be difficult to control the order. For example, achieving a specified total ordering of events across concurrent threads implies interactions across those threads that can be extremely difficult to implement correctly. Research in distributed discrete-event simulation, for example, underscores the subtleties that can arise (see for example [15]).

It is less straightforward to talk about the metric nature of time. Typically, embedded processors have access to external devices (timers) that can be used to measure the passage of time. Programs can poll for the current time, and they can set timers to trigger an interrupt at some time in the future. Using timers in this way implies immediately having to deal with concurrency issues. Interrupt service routines typically preempt currently executing software, and hence conceptually execute concurrently.

Concurrency in software is a challenging issue because the basic software abstraction is not concurrent. The basic abstraction in imperative languages is that the memory of the computer represents the current state of the system, and

instructions transform that state. A program is a sequence of such transformations. The problem with concurrency is that from the perspective of a particular program, the state may change on its own at any time. For example, we could have a sequence of statements:

```
x = 5;
print x;
```

that results in printing the number "6" instead of "5". This could occur, for example, if after execution of the first statement an interrupt occurred, and the interrupt service routine modified the memory location where x was stored. Or it could occur if the computer is also executing a sequence of statements:

```
x = 6;
print x;
```

and a multitasking scheduler happens to interleave the executions of the instructions of the two sequences. Two such sequences of statements are said to be nondeterminate because, by themselves, these two sequences of statements do not specify a single behavior. There is more than one behavior that is consistent with the specification.

Nondeterminism can be desirable in embedded software. Consider for example an embedded system that receives information at random times from two distinct sensors. Suppose that it is the job of the embedded software to fuse the data from these sensors so that their observations are both taken into account. The system as a whole will be nondeterminate since its results will depend on the order in which information from the sensors is processed. Consider the following program fragment:

```
y = getSensorData();    // Block for data
x = 0.9 * x + 0.1 * y;  // Discounted average
print x;                // Display the result
```

This fragment reads data from a sensor and calculates a running average using a discounting strategy, where older data has less effect on the average than newer data.

Suppose that our embedded system uses two threads, one for each sensor, where each thread executes the above sequence of statements repeatedly. The result of the execution will depend on the order in which data arrives from the sensors, so the program is nondeterminate. However, it is also nondeterminate in another way that was probably not intended. Suppose that the multitasking scheduler happens to execute the instructions from the two threads in interleaved order, as shown here:

```
y = getSensorData();     // From thread 1
y = getSensorData();     // From thread 2
x = 0.9 * x + 0.1 * y;   // From thread 1
x = 0.9 * x + 0.1 * y;   // From thread 2
print x;                 // From thread 1
print x;                 // From thread 2
```

The result is clearly not right. The sensor data read by thread 1 is ignored. The discounting is applied twice. The sensor data from thread 2 is counted twice. And the same (erroneous) result is printed twice.

A key capability for preventing such concurrency problems is atomicity. A sequence of instructions is atomic if during the execution of the sequence, no portion of the state that is visible to these instructions changes unless it is changed by the instructions themselves.

Atomicity can be provided by programming languages and/or operating systems through mutual exclusion mechanisms. These mechanisms depend on low-level support for an indivisible test and set. Consider the following modification:

```
acquireLock();          // Block until acquired
y = getSensorData();    // Block for data
x = 0.9 * x + 0.1 * y;  // Discount old value
print x;                // Display the result
releaseLock();          // Release the lock
```

The first statement calls an operating system primitive that tests a shared, boolean-valued variable, and if it is false, sets it to true and returns. If it is true, then it blocks, waiting until it becomes false. It is essential that between the time this primitive tests the variable and the time it sets it to true, that no other instruction in the system can access that variable. That is, the test and set occur as one operation, not as two. The last statement sets the variable to false.

Suppose we now build a system with two threads that each execute this sequence repeatedly to read from two sensors. The resulting system will not exhibit the problem above because the multitasking scheduler cannot interleave the executions of the statements. However, the program is still not correct. For example, it might occur that only one of the two threads ever acquires the lock, and so only one sensor is read. In this case, the program is not fair. Suppose that the multitasking scheduler is forced to be fair, say by requiring it to yield to the other thread each time releaseLock() is called. The program is still not correct, because while one thread is waiting for sensor data, the other thread is blocked by the lock and will fail to notice new data on its sensor. This seemingly trivial problem has become difficult. Rather than trying to fix it within the threading model of computation, we will show that alternative models of computation make this problem easy.

Suppose the program is given by the diagram in figure 1. Suppose that the semantics are those of Kahn process networks (see [21]) augmented with a nondeterministic merge, as done in the YAPI model of computation [4]. In that figure, the components (blocks) are called actors. They have ports (shown by small triangles), with input ports pointing into the blocks and output ports pointing out. Each actor encapsulates functionality that reads input values and produces output values.

In PN semantics, each actor executes continually in its own thread of control. The Sensor1 and Sensor2 actors will produce an output whenever the corresponding sensors have data (this could be done directly by the interrupt service routine,

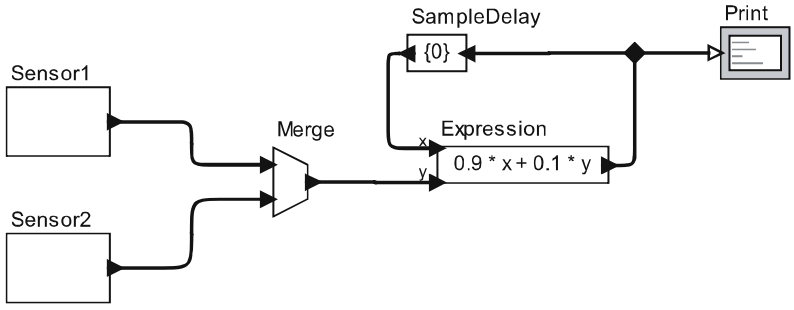

Fig. 1. Process network realization of the sensor fusion example

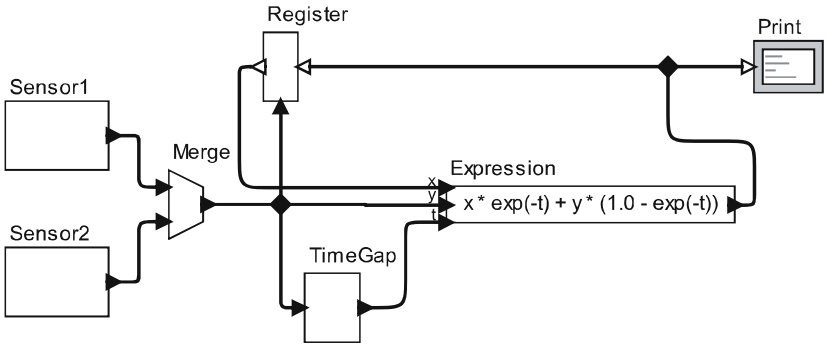

Fig. 2. Discrete event realization of an improved sensor fusion example

for example). The connections between actors represent sequences of data values. The Merge actor will nondeterministically interleave the two sequences at its input ports, preserving the order within each sequence, but yielding arbitrary ordering of data values across sequences. Suppose it is "fair" in the sense that if a data value appears at one of the inputs, then it will "eventually" appear at the output [25]. The remaining actors simply calculate the discounted average and display it. The SampleDelay actor provides an initial "previous average" to work with (which prevents this program from deadlocking for lack of data at the input to the Expression actor). This program exhibits none of the difficulties encountered above with threads, and is both easy to write and easy to understand.

We can now focus on improving its functionality. Notice that the discounting average is not ideal because it does not take into account how old the old data are. That is, there is no time metric. Old data is simply the data previously observed, and there is no measure of how long ago it was read. Suppose that instead of Kahn process networks semantics, we use discrete-event (DE) semantics [18]. A modified diagram is shown in figure 2. In that diagram, the meaning of a connection between actors is slightly different from the meaning of connections in figure 1. In particular, the connection carries a sequence of data values as

before, but each value has a time stamp. The time stamps on any given sequence are nondecreasing. A data value with a time stamp is called an event.

The Sensor1 and Sensor2 actors produce output events stamped with the time at which their respective interrupt service routines are executed. The Merge actor is no longer nondeterministic. Its output is a chronological merge of the two input sequences. The TimeGap actor produces on its output an event with the same time stamp as the input but whose value is the elapsed time between the current event and the previous event (or between the start of execution and the current event if this is the first event). The expression shown in the next actor calculates a better discounted average, one that takes into account the time elapsed. It implements an exponential forgetting function.

The Register actor in figure 2 has somewhat interesting semantics. Its output is produced when it receives a trigger input on the bottom port. The value of the output is that of a previously observed input (or a specified initial value if no input was previously observed). In particular, at any given time stamp, the value of the output does not depend on the value of the input, so this actor breaks what would otherwise be an unresolvable causality loop.

Even with such a simple problem, threaded concurrency is clearly inferior. PN offers a better concurrency model in that the program is easier to construct and to understand. The DE model is even better because it takes into account metric properties of time, which matter in this problem.

In real systems, the contrasts between these approaches is even more dramatic. Consider the following two program fragments:

```
acquireLockA();
acquireLockB();
x = 5;
print x;
releaseLockB();
releaseLockA();
```

and

```
acquireLockB();
acquireLockA();
x = 5;
print x;
releaseLockA();
releaseLockB();
```

If these two programs are executed concurrently in two threads, they could deadlock. Suppose the multitasking scheduler executes the first statement from the first program followed by the first statement from the second program. At this point, the second statement of both programs will block! There is no way out of this. The programs have to be aborted and restarted.

Programmers who use threads have tantalizing simple rules to avoid this problem. For example, "always acquire locks in the same order" [17]. However, this

rule is almost impossible to apply in practice because of the way programs are modularized. Any given program fragment is likely to call methods or procedures that are defined elsewhere, and those methods or procedures may acquire locks. Unless we examine the source code of every procedure we call, we cannot be sure that we have applied this rule.

Deadlock can, of course, occur in PN and DE programs. If in figure 1 we had omitted the SampleDelay actor, or in figure 2 we had omitted the Register actor, the programs would not be able to execute. In both cases, the Expression actor requires new data at all of its input ports in order to execute, and that data would not be able to be provided without executing the Expression actor.

The rules for preventing deadlocks in PN and DE programs are much easier to apply than the rule for threads. For certain models of computation, whether deadlock occurs can be checked through static analysis of the program. This is true of the DE model used above for the improved sensor fusion problem, for example. So, not only was the model of computation more expressive in practice (that is, it more readily expressed the behavior we wanted), but it also had stronger formal properties that enabled static checks that prove the absence of certain flaws (deadlock, in this case).

We will next examine a few of the models of computation that have been used for embedded systems and that form the basis for the work described here.

4 Imperative Concurrent Models

TinyOS has an imperative flavor. What this means is that when one component interacts with another, it gives a command, "do this." The command is implemented as a procedure call. Since this model of computation is also concurrent, we call it an imperative concurrent models of computation.

In contrast, when components in Simulink and SCADE interact, they simply offer data values, "here is some data." It is irrelevant to the component when (or even whether) the destination component reacts to the message. These models of computation have a declarative flavor, since instead of issuing commands, they declare relationships between components that share data. We call such models of computation declarative concurrent models of computation.

TinyOS is a specialized, small-footprint operating system for use on extremely resource-constrained computers, such as 8 bit microcontrollers with small amounts of memory [8]. It is typically used with nesC, a programming language that describes "configurations," which are assemblies of TinyOS components.

A visual rendition of a two-component configuration is shown in figure 3, where the visual notation is that used in [8]. The components are grey boxes with names. Each component has some number of interfaces, some of which it uses and some of which it provides. The interfaces it provides are put on top of the box and the interfaces it uses are put on the bottom. Each interface consists of a number of methods, shown as triangles. The filled triangles represent methods that are called commands and the unfilled triangles represent event handlers. Commands propagate downwards, whereas events propagate upwards.

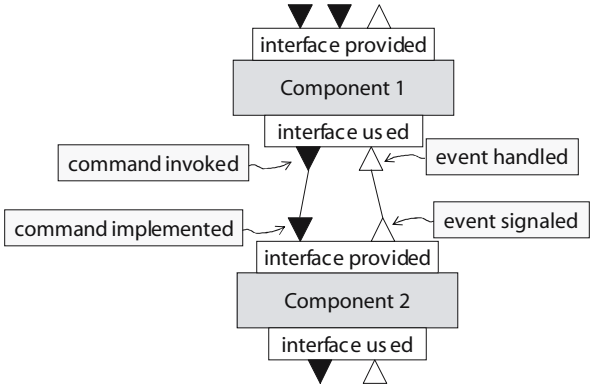

Fig. 3. A representation of a nesC/TinyOS configuration

After initialization, computation typically begins with events. In figure 3, Component 2 might be a thin wrapper for hardware, and the interrupt service routine associated with that hardware would call a procedure in Component 1 that would "signal an event." What it means to signal an event is that a procedure call is made upwards in the diagram via the connections between the unfilled triangles. Component 1 provides an event handler procedure. The event handler can signal an event to another component, passing the event up in the diagram. It can also call a command, downwards in the diagram. A component that provides an interface provides a procedure to implement a command.

Execution of an event handler triggered by an interrupt (and execution of any commands or other event handlers that it calls) may be preempted by another interrupt. This is the principle source of concurrency in the model. It is potentially problematic because event handler procedures may be in the middle of being executed when an interrupt occurs that causes them to begin execution again to handle a new event. Problems are averted through judicious use of the atomic keyword in nesC. Code that is enclosed in an atomic block cannot be interrupted (this is implemented very efficiently by disabling interrupts in the hardware).

Clearly, however, in a real-time system, interrupts should not be disabled for extensive periods of time. In fact, nesC prohibits calling commands or signaling events from within an atomic block. Moreover, no mechanism is provided for an atomic test-and-set, so there is no mechanism besides the atomic keyword for implementing mutual exclusion. The system is a bit like a multithreaded system but with only one mutual exclusion lock. This makes it impossible for the mutual exclusion mechanism to cause deadlock.

Of course, this limited expressiveness means that event handlers cannot perform non-trivial concurrent computation. To regain expressiveness, TinyOS has tasks. An event handler may "post a task." Posted tasks are executed when the machine is idle (no interrupt service routines are being executed). A task may call commands through the interfaces it uses. It is not expected to signal events, however. Once task execution starts, it completes before any other task

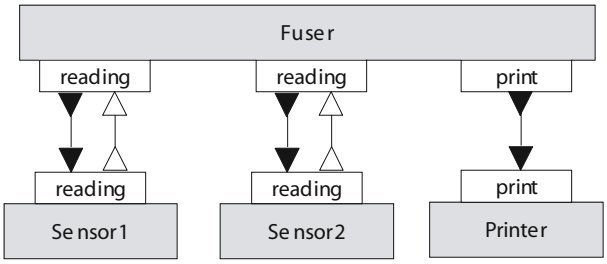

Fig. 4. A sketch of the sensor fusion problem as a nesC/TinyOS configuration

execution is started. That is, task execution is atomic with respect to other tasks. This greatly simplifies the concurrency model, because now variables or resources that are shared across tasks do not require mutual exclusion protocols to protect their accesses. Tasks may be preempted by event handlers, however, so some care must be exercised when shared data is accessed here to avoid race conditions. Interestingly, it is relatively easy to statically analyze a program for potential race conditions [8].

Consider the sensor fusion example from above. A configuration for this is sketched in figure 4. The two sensors have interfaces called "reading" that accept a command a signal an event. The command is used to configure the sensors. The event is signaled when an interrupt from the sensor hardware is handled. Each time such an event is signaled, the Fuser component records the sensor reading and posts a task to update the discounted average. The task will then invoke the command in the print interface of the Printer component to display the result. Because tasks execute atomically with respect to one another, in the order in which they are posted, the only tricky part of this implementation is in recording the sensor data. However, tasks in TinyOS can be passed arguments on the stack, so the sensor data can be recorded there. The management of concurrency becomes extremely simple in this example.

In effect, in nesC/TinyOS, concurrency is much more disciplined than with threads. There is no arbitrary interleaving of code execution, there are no blocking operations to cause deadlock, and there is a very simple mechanism for managing the one nondeterministic preemption that can be caused by interrupts. The price paid for this, however, is that applications must be divided into small, quickly executing procedures to maintain reactivity. Since tasks run to completion, a long-running task will starve all other tasks.

5 Declarative Concurrent Models

Simulink, SCADE, LabVIEW and hardware description languages all have a declarative flavor. The interactions between components are not imperative in that one component does not "tell the other what to do." Instead, a "program" is a declaration of the relationships among components.

Simulink was originally developed as a modeling environment, primarily for control systems. It is rooted in a continuous-time semantics, something that is intrinsically challenging for any software system to emulate. Software is intrinsically discrete, so an execution of a Simulink "program" often amounts to approximating the specified behavior using numerical integration techniques.

A Simulink "program" is an interconnection of blocks where the connections are the "variables," but the value of a variable is a function, not a single value. To complicate things, it is a function defined over a continuum. The Integrator block, for example, takes as input any function of the reals and produces as output the integral of that function. In general, any numerical representation in software of such a function and its integral is an approximation, where the value is represented at discrete points in the continuum. The Simulink execution engine (which is called a "solver") chooses those discrete points using sometimes quite sophisticated methods.

Although initially Simulink focused on simulating continuous dynamics and providing excellent numerical integration, more recently it acquired a discrete capability. Semantically, discrete signals are piecewise-constant continuous-time signals. A piecewise constant signal changes value only at discrete points on the time line. Such signals are intrinsically easier for software, and more precise approximations are possible.

In addition to discrete signals, Simulink has discrete blocks. These have a sampleTime parameter, which specifies the period of a periodic execution. Any output of a discrete block is a piecewise constant signal. Inputs are sampled at multiples of the sampleTime.

Certain arrangements of discrete blocks turn out to be particularly easy to execute. An interconnection of discrete blocks that all have the same sampleTime value, for example, can be efficiently compiled into embedded software. But even blocks with different sampleTime parameters can yield efficient models, when the sampleTime values are related by simple integer multiples.

Fortunately, in the design of control systems (and many other signal processing systems), there is a common design pattern where discrete blocks with harmonically related sampleTime values are commonly used to specify the software of embedded control systems.

Figure 5 shows schematically a typical Simulink model of a control system. There is a portion of the model that is a model of the physical dynamics of the system to be controlled. There is no need, usually, to compile that specification into embedded software. There is another portion of the model that represents a discrete controller. In this example, we have shown a controller that involves multiple values of the sampleTime parameter, shown as numbers below the discrete blocks. This controller is a specification for a program that we wish to execute in an embedded system.

Real-Time Workshop is a product from The MathWorks associated with Simulink. It takes models like that in figure 5 and generates code. Although it will generate code for any model, it is intended principally to be used only on the discrete controller, and indeed, this is where its strengths come through.

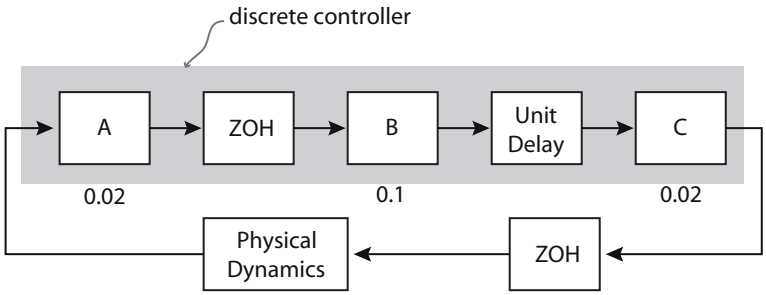

Fig. 5. A representation of a Simulink program

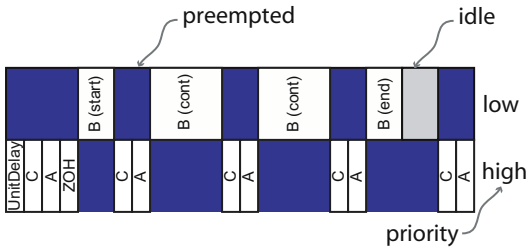

Fig. 6. A simplified representation of a Simulink schedule

The discrete controller shown in figure 5 has fast running components (with sampleTime values of 0.02, or 20 ms) and slow running components (with sampleTime values of 0.1, or 1/10 of a second). In such situations, it is not unusual for the slow running components to involve much heavier computational loads than the fast running components. It would not do to schedule these computations to execute atomically, as is done in TinyOS and Click (and SCADE). This would permit the slow running component to interfere with the responsivity (and time correctness) of the fast running components.

Simulink with Real-Time Workshop uses a clever technique to circumvent this problem. The technique exploits an underlying multitasking operating system with preemptive priority-driven multitasking. The slow running blocks are executed in a separate thread from the fast running blocks, as shown in figure 6. The thread for the fast running blocks is given higher priority than that for the slow running blocks, ensuring that the slow running code cannot block the fast running code. So far, this just follows the principles of rate-monotonic scheduling [23]. But the situation is a bit more subtle than this, because data flows across the rate boundaries. Recall that Simulink signals have continuous-time semantics, and that discrete signals are piecewise constant. The slow running blocks should "see" at their input a piecewise constant signal that changes values at the slow rate.To guarantee that, the model builder is required to put a zero-order hold (ZOH) block at the point of the rate conversion. Failure to do so will trigger an error message. Cleverly, the code for the ZOH runs at the rate of the slow

block but at the priority of the fast block. This makes it completely unnecessary to do semaphore synchronization when exchanging data across these threads.

When rate conversions go the other way, from slow blocks to fast blocks, the designer is required to put a UnitDelay block, as shown in figure 5. This is because the execution of the slow block will typically stretch over several executions of the fast block, as shown in figure 6.

To ensure determinacy, the updated output of the block must be delayed by the worst case, which will occur if the execution stretches over all executions of the fast block in one period of the slow block. The unit delay gives the software the slack it needs in order to be able to permit the execution of the slow block to stretch over several executions of the fast one. The UnitDelay executes at the rate of the slow block but at the priority of the fast block.

This same principle has been exploited in Giotto [12], which constrains the program to always obey this multirate semantics and provides (implicitly) a unit delay on every connection. In exchange for these constraints, Giotto achieves strong formal structure, which results in, among other things, an ability to perform schedulability analysis (the determination of whether the specified real-time behavior can be achieved by the software).

The Simulink model has weaknesses, however. The sensor fusion problem that we posed earlier does not match its discrete multitasking model very well. While it would be straightforward to construct a discrete multitasking model that polls the sensors at regular (harmonic) rates, reacting to stimulus from the sensors at random times does not fit the semantics very well. The merge shown in figure 2 would be challenging to accomplish in Simulink, and it would not benefit much from the clever code generation techniques of Real-Time Workshop.

In figure 2, we give a discrete-event model of an improved sensor fusion algorithm with an exponential forgetting function. Discrete-event modeling is widely used in electronic circuit design (VHDL and Verilog are discrete-event languages), computer network modeling and simulation (OPNET Modeler and Ns-2, for example), and many other disciplines. In discrete-event models, the components interact via signals that consist of events, which typically carry both a data payload and a time stamp. A straightforward execution of these models uses a centralized event queue, where events are sorted by time stamp, and a runtime scheduler dispatches events to be processed in time order. Compared to the Simulink/RTW model, there is much more flexibility in DE because discrete execution does not need to be periodic. This feature is exploited in the model of figure 2, where the Merge block has no simple counterpart in Simulink.

A great deal of work has been done on efficient and distributed execution of such models, much of this work originating in either the so-called "conservative" technique of Chandy and Misra [3] or the speculative execution methods of Jefferson [15]. More interesting is the work in the Croquet Project, which focuses on optimistic techniques in the face of unreliable components. Croquet has principally been applied to three-D shared immersion environments on the internet, similar to the ones that might be used in interactive networked gaming. Much less work has been done in adapting these models as an execution platform for

embedded software, but there is some early work that bears a strong semantic resemblance to DE modeling techniques [24][9]. A significant challenge is to achieve the timed semantics efficiently while building on software abstractions that have abstracted away time.

6 Discrete-Event Runtime Framework

The ability of TinyOS and nesC to create thin wrappers around hardware provides a simple and understandable mechanism for creating event-triggered, fine-grained, atomic reactions to external events. When these external events trigger significant computations, nesC programs will "post tasks" that are executed later. These tasks, however, all execute atomically with respect to one another, and hence a long-running task will block other tasks. This can create unacceptable latencies, and often forces software designers to manually divide long-running tasks into more fine-grain ones.

Simulink and Giotto, by contrast, freely mix long-running tasks with hard-real-time fine-grained tasks by exploiting the properties of an underlying priority-driven multitasking real-time operating system. They do this without requiring programmers to specify priorities or use mutexes or semaphores. However, these tasks are required to be periodic, and their latencies are strongly related to their periods, so they lack the event-triggered, reactive nature of nesC programs.

These two ideas can be combined within a dataflow framework with elements borrowed from discrete-event models to specify timing properties. Dependencies within the dataflow model can be statically analyzed, and with a carefully chosen variant of dataflow called heterochronous dataflow (HDF) [10], schedulability becomes decidable and synthesis of efficient embedded software becomes possible. We believe that the resulting language will prove expressive, efficient, understandable, and analyzable.

We are building a prototype of this combination of HDF and DE using the Ptolemy II framework [7]. This prototype can synthesize multitasking C code for execution on embedded processors or general-purpose processors. That is, the target language for the compiler will be C. The source language will be graphical, exploiting the graphical syntaxes supported by Ptolemy II. We will specifically target instrumentation applications, and, at coarser temporal granularity, distributed games. We leverage a C code generator for Ptolemy II that supports HDF [10], built by Jackie Mankit Leung and Gang Zhou, for code generation.

The overall architecture of an application is a distributed discrete-event model of interactions of concurrent real-time components (which we call actors). The components themselves have functionality that can be specified either by dataflow models, combinations of dataflow and state machines (heterochronous dataflow), or conventional programming languages (C or Java, in this case).

Discrete-event semantics is typically used for modeling physical systems where atomic events occur on a time line. For example, hardware description languages for digital logic design, such as Verilog and VHDL, are discrete-event languages.

So are many network modeling languages, such as OPNET Modeler[1] and Ns-2[2]. Our approach is not to model physical phenomena, but rather to specify coordinated real-time events to be realized in software. Execution of the software will first obey discrete-event semantics, just as done in DE simulators, but it will do so with specified real-time constraints on certain actions. Our technique is properly viewed as providing a semantic notion of model time together with a relation between the model time of certain events and their physical time.

Our premise is that since DE models are natural for modeling real-time systems, they should be equally natural for specifying real-time systems. Moreover, we can exploit their formal properties to ensure determinism in ways that evades many real-time software techniques. Network time synchronization makes it possible for discrete-event models to have a coherent semantics across distributed nodes. Just as with distributed DE simulation, it will be neither practical nor efficient to use a centralized event queue to sort events in time order. Our goal will be to compile DE models for efficient and predictable distributed execution.

We emphasize that while distributed execution of DE models has long been used to exploit parallel computation to accelerate simulation [31], we are not interested in accelerated simulation. Instead, we are interested in systems that are intrinsically distributed. Consider factory automation, for example, where sensors and actuators are spread out physically over hundreds of meters. Multiple controllers must coordinate their actions over networks. This is not about speed of execution but rather about timing precision. We use the global notion of time that is intrinsic in DE models as a binding coordination agent.

For accelerated simulation, there is a rich history of techniques. So-called "conservative" techniques advance model time to t only when each node can be assured that they have seen all events time stamped t or earlier. For example, in the well-known Chandy and Misra technique [3], extra messages are used for one execution node to notify another that there are no such earlier events. For our purposes, this technique binds the execution at the nodes too tightly, making it very difficult to meet realistic real-time constraints.

So-called "optimistic" techniques perform speculative execution and backtrack when the speculation is incorrect [15]. Such optimistic techniques will also not work in our context, since backtracking physical interactions is not possible.

Our method is called PTIDES, Programming Temporally Integrated Distributed Embedded Systems [32]. It is conservative, in the sense that events are processed only when we are sure it is safe to do so. But we achieve significantly looser coupling than Chandy and Misra using a new method that we call *relevant dependency analysis*. We leverage the concept of causality interfaces introduced in [22], adapting these interfaces to distributed discrete-event models. We have developed the concept of "relevant dependency" to formally capture the ordering constraints of temporally ordered events that have a dependency relationship. This formal structure provides an algebra within which we can perform schedulability analysis of distributed discrete-event models.

[1] http://opnet.com/products/modeler/home.html
[2] http://www.isi.edu/nsnam/ns

Our emphasis is on efficient distributed real-time execution. Our framework uses model time to define execution semantics, and constraints that bind certain model time events to physical time. A correct execution will simply obey the ordering constraints implied by model time and meet the constraints on events that are bound to physical time.

6.1 Motivating Example

We motivate our programming model by considering a simple distributed real-time application. Suppose that at two distinct machines A and B we need to generate precisely timed physical events under the control of software. Moreover, the devices that generate these physical events respond after generating the event with some data, for example sensor data. We model this functionality with an actor that has one input port and one output port, depicted graphically as follows:

This actor is a software component that wraps interactions with device drivers. We assume that it does not communicate with any other software component except via its ports. At its input port, it receives a potentially infinite sequence of time-stamped values, called events, in chronological order. The sequence of events is called a signal. The output port produces a time-stamped value for each input event, where the time stamp is strictly greater than that of the input event. The time stamps are values of model time. This software component binds model time to physical time by producing some physical action at the real-time corresponding to the model time of each input event. Thus, the key real-time constraint is that input events must be made available for this software component to process them at a physical time strictly earlier than the time stamp. Otherwise, the component would not be able to produce the physical action at the designated time.

Figure 7 shows a distributed DE model to be executed on a two-machine, time-synchronized platform. The dashed boxes divide the model into two parts, one to be executed on each machine. The parts communicate via signal s_1. We assume that events in this signal are sent over a standard network as time-stamped values.

The Clock actors in the figure produce time-stamped outputs where the time stamp is some integer multiple of a period p (the period can be different for each clock). Upon receiving an input with time stamp t, the clock actor will produce an output with time stamp np where n is the smallest integer so that $np \geq t$. There are no real-time constraints on the inputs or outputs of these actors.

The Merge actor has two input ports. It merges the signals on the two input ports in chronological order (perhaps giving priority to one port if input events

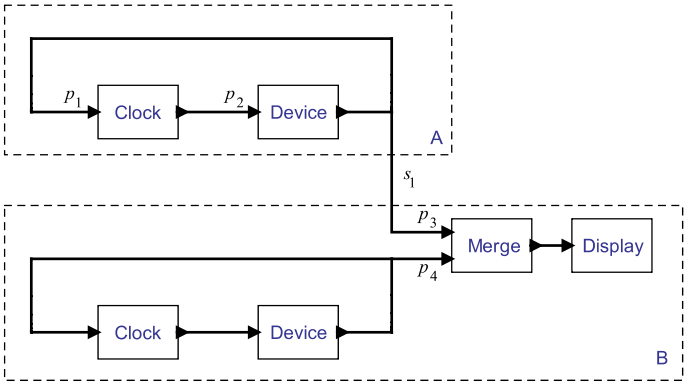

Fig. 7. A simple distributed instrumentation example

have identical time stamps). A conservative implementation of this Merge requires that no output with time stamp t be produced until we are sure we have seen all inputs with time stamps less than or equal to t. There are no real-time constraints on the input or output events of the Merge actor.

The Display actor receives input events in chronological (time-stamped) order and displays them. It also has no real-time constraints.

A brute-force implementation of a conservative distributed DE execution of this model would stall execution in platform B at some time stamp t until an event with time stamp t or larger has been seen on signal s_1. Were we to use the Chandy and Misra approach, we would insert null events into s_1 to minimize the real-time delay of these stalls. However, we have real-time constraints at the Device actors that will not be met if we use this brute-force technique. Moreover, it is intuitively obvious that such a conservative technique is not necessary. Since the actors communicate only through their ports, there is no risk in processing events in the upper Clock-Device loop ahead of time stamps received on s_1. Our PTIDES technique formalizes this observation using causality analysis.

To make this example more concrete, we have in our lab prototype systems provided by Agilent that implement IEEE 1588. These platforms include a Linux host and simple timing-precise I/O hardware. Specifically, they include a device driver API where the software can request that the hardware generate a digital clock edge (a voltage level change) at a specified time. After generating this level change, the hardware interrupts the processor, which resets the level to its original value. Our implementation of the Device actor takes input events as specification of when to produce these level changes. That is, it produces a rising edge at physical time equal to the model time of an input event. After receiving an input, it outputs an event with time stamp equal to the physical time at which the level is restored to its original value. Thus, its input time stamps must precede physical time, and its output events are guaranteed to follow physical time. This physical setup makes it easy to measure precisely the real-time behavior of the system (oscilloscope probes on the digital I/O connectors tell it all).

The feedback loops around the two Clock and Device actors ensure that the Device does not get overwhelmed with requests for future level changes. It may not be able to buffer those requests, or it may have a finite buffer. Without the feedback loop, since the ports of the Clock actor have no real-time constraints, there would be nothing to keep it from producing output events much faster than real time.

This model is an abstraction of many realistic applications. For example, consider two networked computers controlling cameras pointing at the same scene from different angles. Precise time synchronization allows them to take sequences of pictures simultaneously. Merging two synchronous pictures creates a 4D view for the scene (three physical dimensions and one time).

PTIDES programs are discrete-event models constructed as networks of actors, as in the example above. For each actor, we specify a physical host to execute the actor. We also designate a subset of the input ports to be *real-time ports*. Time-stamped events must be delivered to these ports before physical-time exceeds the time stamp. Each real-time port can optionally also specify a *setup time* τ, in which case it requires that each input event with time stamp t be received before physical time reaches $t - \tau$. A model is said to be *deployable* if these constraints can be met for all real-time ports. Causality analysis can reveal whether a model is deployable.

The key idea is that events only need to be processed in time-stamp order when they are causally related. We defined formal interfaces to actors that tells us when such causal relationships exist.

6.2 Summary of Relevant Dependency Analysis

A formal framework for analyzing causality relationships to determine the minimal ordering constraints on processing events is given in [32]. We give a summary of the key results here. The technique is based on causality interfaces [22], which provide a mechanism that allows us to analyze delay relationships among actors. The interface of actors contains ports on which actors receive or produce events. Each port is associated with a signal. A *causality interface* declares the dependency that output events have on input events.

A program is given as a composition of actors, by which we mean a set of actors and connectors linking their ports. Given a composition and the causality interface of each actor, we can determine the dependencies between any two ports in the composition. However, these dependencies between ports do not tell the whole story. Consider the Merge actor in figure 7. It has two input ports, and the dependency analysis tells that there is no path between these ports. However, these ports have an important relationship, noted above. In particular, the Merge actor cannot react to an event at one port with time stamp t until it is sure it has seen all events at the other port with time stamp less than or equal to t. This fact is not captured in the dependencies. To capture it, we use *relevant dependencies*. Based on the causality interface of actors, the *relevant dependency* on any pair (p_1, p_2) of input ports specifies whether an event at p_1 will affect an output signal that may also depend on an event at p_2. Given the

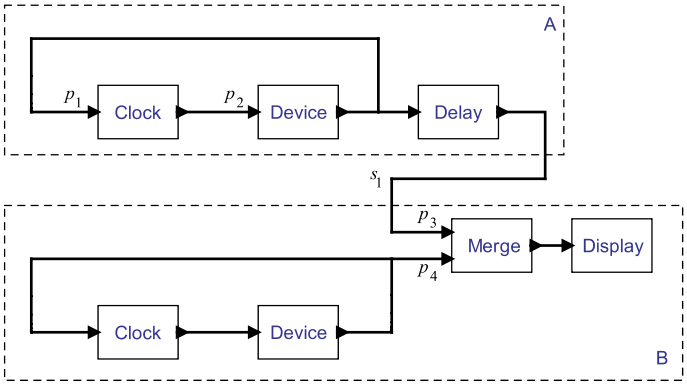

Fig. 8. The motivating example with a delay actor

relevant dependency interfaces for all actors in a composition, we can establish the relevant dependency between any two input ports in the composition.

When the relevant dependency from input port p_1 to p_2 is r, $r \in \mathbb{R}_0$, this means that any event with time stamp t_2 at p_2 can be processed when all events at p_1 are known up to time stamp $t_2 - r$. When the relevant dependency from p_1 to p_2 is ∞, this means that events at p_2 can be processed without knowing anything about events at p_1.

What we gain from the dependency analysis is that we can specify which events can be processed out of order, and which events have to be processed in order. Recall that p_2 is designated as a real-time port. Relevant dependency analysis tells us that events at p_2 can be processed without knowing anything about events at p_3 or p_4. This is the first result we were after. It means that the arrival events over the network into p_3 need not interfere with meeting real-time constraints at p_2. This would not be achieved with a Chandy and Misra policy. And unlike optimistic policies, there will never be any need to backtrack.

If we modify the model in figure 7 by adding a Delay actor with a delay parameter d, we get a new model as shown in figure 8. Relevant dependency analysis now tells us that an event with time stamp t at p_4 can be processed if all events with time stamps smaller than or equal to $t - d$ at p_3 have been processed. With the same assumptions as discussed in section 6.1 (an event with model time t is produced at physical time t by the Device process, and the network delay is bounded by C), at physical time $t - d + C$ we are sure that we have seen all events with time stamps smaller than $t - d$ at p_3. Hence, an event e at p_4 with time stamp t can be processed at physical time $t - d + C$ or later. Note that although the Delay actor has no real-time properties at all (it simply manipulates model time), its presence loosens the constraints on the execution.

In [32] we show that relevant dependencies induce a partial order (called the *relevant order*) on events. We use notation $<_r$ for the relevant order. We interpret $e_1 <_r e_2$ to mean that e_1 must be processed before e_2. Two events e_1 and e_2 are not comparable, denoted as $e_1 || _r e_2$, if neither $e_1 <_r e_2$, nor $e_2 <_r e_1$. If $e_1 || _r e_2$,

then e_1, e_2 can be processed in any order. What we mean by "processed" is that the actor that is the destination of the event is *fired*, meaning that it is executed and allowed to react to the event. It is then straightforward to show that any execution that respects the relevant order correctly implements discrete-event semantics.

We further show in [32] that this technique can be adapted to distributed execution if we are given bounds on the communication latency and on the timing synchronization order.

7 Conclusion

Existing methods for addressing real-time computation typically deal with a portion of the problem of constructing and executing real-time programs. Real-time operating systems (RTOSs) provide mechanisms for prioritizing tasks and triggering computations in response to timer interrupts. Time-triggered networking techniques such as the Time Triggered Architecture (TTA) provide deterministic sharing of networking resources and insulation from faults. Network time synchronization protocols such as NTP and IEEE 1588 provide a common time base across computers on a network. All of these technologies, however, are used with relatively conventional concurrency models (threads and processes) and conventional programming languages. This paper elevates timing and distribution to the level of the programmers model, so that applications are built by directly expressing timing and distribution properties. The objective is a framework for designing deployable timed distributed systems. Our technique adapts discrete-event semantics, traditionally used for modeling and simulation, for use as a programmers' model for distributed real-time software.

Acknowledgements

This paper describes work that is part of the Ptolemy project, which is supported by the National Science Foundation (NSF award number CCR-00225610), and Chess (the Center for Hybrid and Embedded Software Systems), which receives support from NSF, the State of California Micro Program, and the following companies: Agilent, DGIST, General Motors, Hewlett Packard, Infineon, Microsoft, and Toyota.

References

1. O. Avissar, R. Barua, and D. Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *Trans. on Embedded Computing Sys.*, 1(1):6–26, 2002.
2. G. Berry. The effectiveness of synchronous languages for the development of safety-critical systems. White paper, Esterel Technologies, 2003.
3. K. M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Trans. on Software Engineering*, 5(5), 1979.

4. E. A. de Kock, G. Essink, W. J. M. Smits, P. van der Wolf, J.-Y. Brunel, W. Kruijtzer, P. Lieverse, and K. A. Vissers. YAPI: Application modeling for signal processing systems. In *37th Design Automation Conference (DAC'00)*, pages 402–405, Los Angeles, CA, 2000.
5. E. Dijkstra. Cooperating sequential processes. In E. F. Genuys, editor, *Programming Languages*. Academic Press, 1968.
6. A. Dominguez, S. Udayakumaran, and R. Barua. Heap data allocation to scratch-pad memory in embedded systems. *Journal of Embedded Computing*, 1(4), 2005.
7. 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.
8. D. Gay, P. Levis, R. v. Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. In *Programming Language Design and Implementation (PLDI)*, 2003.
9. A. Ghosal, T. A. Henzinger, C. M. Kirsch, and M. A. Sanvido. Event-driven programming with logical execution times. In *Seventh International Workshop on Hybrid Systems: Computation and Control (HSCC)*, volume Lecture Notes in Computer Science 2993, pages 357–371. Springer-Verlag, 2004.
10. A. Girault, B. Lee, and E. A. Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions On Computer-aided Design Of Integrated Circuits And Systems*, 18(6):742–760, 1999.
11. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1319, 1991.
12. T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: A time-triggered language for embedded programming. In *EMSOFT 2001*, volume LNCS 2211, Tahoe City, CA, 2001. Springer-Verlag.
13. J. Hill, R. Szewcyk, A. Woo, D. Culler, S. Hollar, and K. Pister. System architecture directions for networked sensors. In *9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 93–104, 2000.
14. C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.
15. D. Jefferson. Virtual time. *ACM Trans. Programming Languages and Systems*, 7(3):404–425, 1985.
16. E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, 2000.
17. D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, Reading MA, 1997.
18. E. A. Lee. Modeling concurrent real-time processes using discrete events. *Annals of Software Engineering*, 7:25–45, 1999.
19. E. A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.
20. E. A. Lee, S. Neuendorffer, and M. J. Wirthlin. Actor-oriented design of embedded hardware and software systems. *Journal of Circuits, Systems, and Computers*, 12(3):231–260, 2003.
21. E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, 1995.
22. E. A. Lee, H. Zheng, and Y. Zhou. Causality interfaces and compositional causality analysis. In *Foundations of Interface Technologies (FIT), Satellite to CONCUR*, San Francisco, CA, 2005.
23. C. L. Liu and J. W. Leyland. Scheduling algorithms for multiprogramming in a hard real time environment. *Journal of the ACM*, 20(1):46–61, 1973.

24. J. Liu and E. A. Lee. Timed multitasking for real-time embedded software. *IEEE Control Systems Magazine*, pages 65–75, 2003.
25. P. Panangaden and V. Shanbhogue. The expressive power of indeterminate dataflow primitives. *Information and Computation*, 98(1):99–131, 1992.
26. T. M. Parks. A comparison of MPI and process networks. In *International Parallel and Distributed Processing Symposium, Workshop on Java for Parallel and Distributed Computing*, Denver, CO, 2005.
27. J. L. Pino, S. Ha, E. A. Lee, and J. T. Buck. Software synthesis for dsp using Ptolemy. *Journal on VLSI Signal Processing*, 9(1):7–21, 1995.
28. D. A. Smith, A. Kay, A. Raab, and D. P. Reed. Croquet: A collaboration system architecture. White paper, 2003.
29. J. A. Stankovic. Misconceptions about real-time computing: a serious problem for next-generation systems. *Computer*, 21(10):10–19, 1988.
30. N. Wirth. Toward a discipline of real-time programming. *Communications of the ACM*, 20(8):577–583, 1977.
31. B. P. Zeigler, H. Praehofer, and T. G. Kim. *Theory of Modeling and Simulation*. Academic Press, 2nd edition, 2000.
32. Y. Zhao, E. A. Lee, and J. Liu. Programming temporally integrated distributed embedded systems. Technical Report UCB/EECS-2006-82, EECS Department, University of California, Berkeley, May 28 2006.