CONCURRENT MODELS OF COMPUTATION FOR EMBEDDED SOFTWARE

by

Edward Lee and Stephen Neuendorffer

Memorandum No. UCB/ERL M04/26

22 July 2004

CONCURRENT MODELS OF COMPUTATION FOR EMBEDDED SOFTWARE

by

Edward Lee and Stephen Neuendorffer

Memorandum No. UCB/ERL M04/26

22 July 2004

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Concurrent Models of Computation for Embedded Software

Edward Lee and Stephen Neuendorffer
Memorandum No. UCB/ERL M04/26, July 22, 2004
EECS Department, University of California at Berkeley
Berkeley, CA 94720, U.S.A.

Abstract

The prevailing abstractions for software are better suited to the traditional problem of computation, namely transformation of data, than to the problems of embedded software. These abstractions have weak notions of concurrency and the passage of time, which are key elements of embedded software. Innovations such as nesC/TinyOS (developed for programming very small programmable sensor nodes called "motes"), Click (created to support the design of software-based network routers), Simulink with Real-Time Workshop (created for embedded control software), and Lustre/SCADE (created for safety-critical embedded software) offer abstractions that address some of these issues and differ significantly from the prevailing abstractions in software engineering. This paper surveys some of the abstractions that have been explored.

1 Introduction

Embedded software has traditionally been thought of as "software on small computers." In this traditional view, the principal problem is resource limitations (small memory, small data word sizes, and relatively slow clocks). The solutions emphasize efficiency; software is written at a very low level (in assembly code or C), operating systems with a rich suite of services are avoided, and specialized computer architectures such as programmable DSPs and network processors are developed to 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.

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. Why then has embedded software design and development changed so little? It may be because extreme competitive pressure in products based on embedded software, such as consumer electronics, rewards only the most efficient solutions. This argument is questionable, however, since there are many examples where functionality has proven more important than efficiency. We will argue that resource limitations are not the only defining factor for embedded software, and may not even be the principal factor now that the technology has improved so much.

Resource limitations are an issue to some degree with almost all software. So generic improvements in software engineering should, in theory, also help with embedded software. There are several hints, however, that embedded software is different in more fundamental ways. For one, object-oriented techniques such as inheritance, dynamic binding, and polymorphism are rarely used in practice with embedded software development. In another example, processors used for embedded systems often avoid the memory hierarchy techniques that are used in general purpose processors to deliver large virtual memory spaces and faster execution using caches. In a third example, automated memory management, with allocation, deallocation, and garbage collection, are largely avoided in embedded software. To be fair, there are some successful applications of these technologies in embedded software, such as the use of Java in cell phones, but their application remains limited and is largely confined to providing the services in embedded systems that are actually more akin with general purpose software applications (such as database services in cell phones).

There are further hints that the software solutions for embedded software may ultimately differ significantly from those for general purpose software. We point to four recent cases where fundamentally different software design techniques have been applied to embedded software. All four define concurrency models, component architectures, and management of time-critical operations in ways that are

[0]This paper describes work that is part of the Ptolemy project, which is supported by the National Science Foundation (NSF award number CCR-00225610), the Defense Advanced Research Projects Agency (DARPA), and Chess (the Center for Hybrid and Embedded Software Systems), which receives support from NSF and the following companies: Daimler-Chrysler, Hitachi, Honeywell, Toyota and Wind River Systems.

significantly different from prevailing software engineering techniques. The first two are nesC with TinyOS [24, 17], which was developed for programming very small programmable sensor nodes called "motes," and Click [32, 31], 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 next two are Simulink with Real-Time Workshop (from The MathWorks), which was created for embedded control software and is widely used in the automotive industry, and SCADE (from Esterel Technologies, see [4]), which was created for safety-critical embedded software and is used in avionics. These next two have a more declarative flavor, where components interact principally through messages rather than procedure calls. There are quite a few more examples that we will discuss below. The amount of experimentation with alternative models of computation for embedded software is yet a further indication that the prevailing software abstractions are inadequate.

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.

Software abstracts away time, replacing it with ordering. In the prevailing software abstraction, that of 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 abstraction, that of threads or processes,[1] typically provided by the operating system, but occasionally by the language (as in Java).

We will argue that the lack of timing in the core abstraction is a flaw, from the perspective of embedded software, and that threads as a concurrency model are a poor match to embedded systems. They are mainly focused on providing an illusion of concurrency 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.

Of the four cases cited above, not one uses threads as the concurrency model. Of the four, only one (Simulink) is explicit about timing. This may be a reflection of how difficult it is to be explicit about timing when the most basic notion of computation has abstracted time away. To be fair, the others do provide mechanisms to managing time-critical events. TinyOS and Click both provide access to hardware timers, but this access is largely orthogonal to the semantics. It is treated as an I/O interaction.

There are, of course, software abstractions that admit concurrency without resorting to threads. In functional languages (see [27] for example), programs are compositions of declarative relationships, not specifications of an order of operations. But although declarative techniques have been used in embedded software (e.g., Simulink and SCADE), functional languages have found almost no usage in embedded software. Thus, whether a language is imperative or declarative probably has little bearing on whether it is useful for embedded software.

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). We argue that the prevailing concurrency model based on threads does not achieve adequate reliability. 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. These techniques often lead to deadlock or livelock conditions, where all or part of a program cannot continue executing. In general purpose computing, this is inconvenient, and typically forces a restart of the program (or even a reboot of the machine). 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 in deployed form 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.

It can be argued that the unreliability of multi-threaded 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. For example, development schedule extensions are as likely to degrade the reliability of programs that are difficult to understand as they are to improve it.

Formal methods can help detect flaws in threaded programs, and in the process can improve the understanding

[1] Threads are processes that can share data. The distinction between the two is not important in our discussion, so we will use the term "threads" generically to refer to both.

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.

All four of the cases cited above offer concurrency models that are much easier to understand than threads that interact via semaphores and mutual exclusion locks.

Simulink and SCADE are based on a synchronous abstraction, where components conceptually execute simultaneously, aligned with one or more interlocked clocks. SCADE relies on an abstraction where components appears to execute instantaneously, whereas Simulink is more explicit about the passage of time and supports definition of tasks that take time to execute and execute concurrently with other tasks. In both cases, every (correctly) compiled version of the program will execute identically, in that if it is given the same inputs, it will produce the same outputs. In particular, the execution does not depend on extraneous factors such as processor speed. Even this modest objective is often hard to achieve using threads directly.

TinyOS and Click offer concurrency models that are closer to the prevailing software abstractions, since they rely on procedure calls as the principle component interaction mechanism. However, neither model includes threads. The key consequence is that a programmer can rely on the atomicity of the execution of most program segments, and hence does not usually need to explicitly deal with mutual exclusion locks or semaphores. The result again is more comprehensible concurrent programs.

2 Concurrency and Time

In embedded software, concurrency and time are essential aspects of a design. In this section, we outline the potential problems that software faces in dealing with these aspects.

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 [12, 28]).

It is less straightforward to talk about the metric nature of time. Typically, embedded processors have access to external devices called 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, I 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 is 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[2] 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 leave this an exercise), we will show that alternative models of computation make this problem easy.

Suppose that the program is given by the diagram in figure 1.[3] Suppose that the semantics are those of Kahn process networks (PN) [30, 37] augmented with a nondeterministic merge [1, 15]. 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, 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[45]. 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 is. That is, there is

[2]Mutual exclusion locks may also be provided as part of a programming language. The "synchronized" keyword in Java, for example, performs the same function as our "acquireLock" command.

[3]We give this program using a visual syntax to emphasize its concurrent semantics, and because visual syntaxes are commonly used for languages with similar semantics, such as SCADE and Simulink. But the visual syntax makes this no less a program.

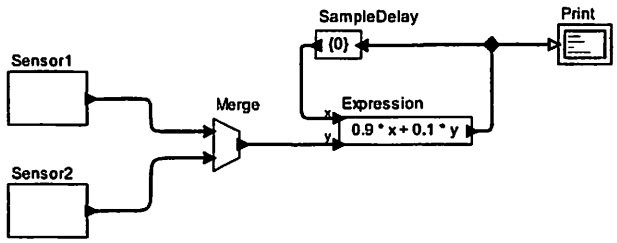

Figure 1. Process network realization of the sensor fusion example.

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 [12, 35]. 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.[4] 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.

[4] A minor detail is that we have to decide how to handle simultaneous input events. We could, for example, produce them both at the output with the one from the top input port preceding the one at the bottom input port. The semantics of simultaneous events is considered in detail in [35].

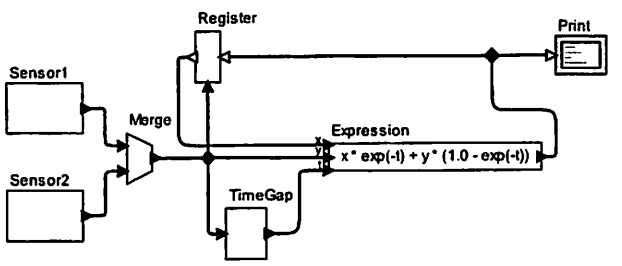

Figure 2. Discrete event realization of an improved sensor fusion example.

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" [33]. 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.[5]

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

[5] In principle, it might be possible to devise a programming language where the locks that are acquired by a procedure are part of the type signature of the procedure, much as in Java where the exceptions that are thrown by a procedure are part of its type signature. However, we know of no language that does this.

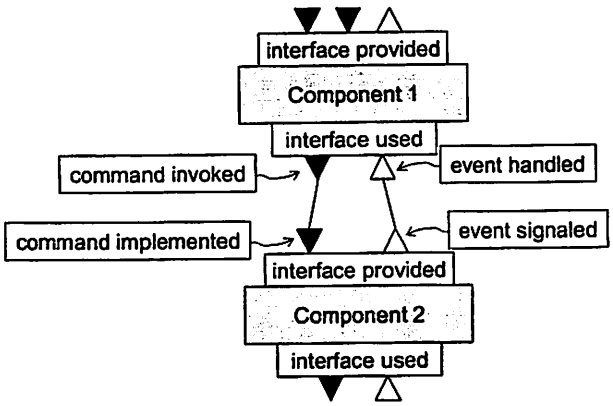

Figure 3. A representation of a nesC/TinyOS configuration.

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.

3 Imperative Concurrent Models

As mentioned above, TinyOS and Click have 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 these models of computation are also concurrent, we call them *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.

We begin with the imperative concurrent models of computation.

3.1 nesC/TinyOS

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[24]. It is typically used with nesC, a programming language that describes "configurations," which are assemblies of TinyOS components[17].

A visual rendition of a two-component configuration is shown in figure 3, where the visual notation is that used in [17]. 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.

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 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

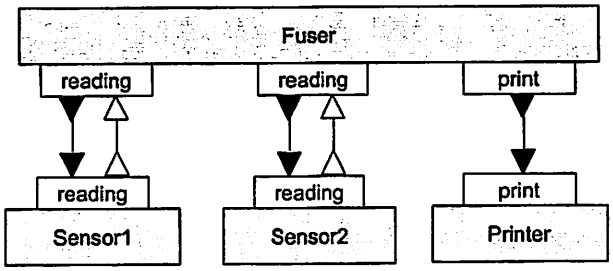

Figure 4. An sketch of the sensor fusion problem as a nesC/TinyOS configuration.

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 [17].

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 fig:Simulink 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.

3.2 Click

Click was originally developed for designing software implementations of network routers on general purpose computers running Linux [32, 31]. It has been recently adapted for designing software for specialized network processors [47], and has proven to offer effective abstractions for this style of embedded software, at least. The abstractions have a great deal of potential for any embedded software that deals with multiple converging asynchronous streams of stimuli.

As with nesC/TinyOS, in the Click model, connections between components represent method bindings. Click does not have the bidirectional interfaces of TinyOS, but it has its own twist that can be used to accomplish similar objectives. In Click, connections between ports can be *push* or *pull*. In a push connection, the method call originates with the source of the data. That is, the producer component calls the consumer component. In a pull connection, the method call originates with the consumer. That is, the consumer component calls the producer component to demand data. It is worth noting that there are middleware frameworks with similar push/pull semantics, such as the CORBA event service [44, 40]. These, however, are aimed at distributed computation rather than at managing concurrency within a single CPU. Click executes in a single thread, and we will see that this simplifies the design of Click applications compared to what would be required by distributed models.

Figure 5 shows a Click model using the visual notation from [31]. Boxes again represent components, and ports are shown either as rectangles (for output ports) or triangles (for input ports). If a port is filled in black, then it is required to link to a push connection. If it is filled in white, then it is required to link to a pull connection. If it has a double outline, then it is agnostic, and can be linked to either type of connection.

A component with just a push output port, like Sensor1 and Sensor2 in figure 5, can function as a thin wrapper around hardware that will produce data. Conceptually, the component autonomously [6] initiates a reaction by pushing data on its output port, which means calling a method in a downstream component. That method in the downstream component may itself trigger further reactions by either pushing data to output ports or pulling data from input ports.

[6]Currently, Click accomplishes this by repeatedly executing a task that polls the hardware, instead of an interrupt service routine, but it does not seem hard to adapt the model leverage interrupt service routines, if desired.

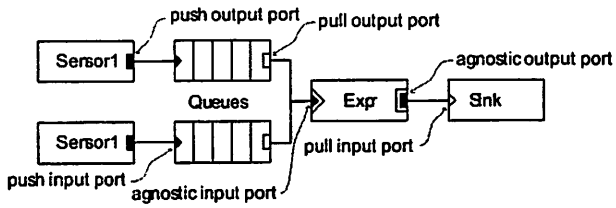

Figure 5. A representation of a Click program.

In the example shown in figure 5, the components downstream of Sensor1 and Sensor2 are Queues. They have push inputs and pull outputs. When a method is called to push data into them, that method simply stores the data on a queue. When a method is called to pull data from their outputs, either a datum is provided or a null value is provided to indicate that no data are available.

Click runs in a single thread, so the push and pull methods of the queue component will be atomic with respect to one another. Thus, no special care needs to be taken to manage the fact that callers from the left and from the right will both access the same (queue) data structure.

In the example shown in figure 5, the Sink component has a single pull input. This component might, for example, have a task on the Click task queue that is repeatedly executed and pulls data from the input port. The upstream component, labeled Expr, has agnostic input and output ports. Because of the way it is connected, these ports will be used as pull ports. A pull from the Sink will cause the Expr component to pull data from the queues. Notice that it can implement a scheduling strategy (such as round robin) to access the queues fairly.

It is easy to see how the example in figure 5 could be adapted to implement the sensor fusion problem. Once again, the representation is simple and clear, with no particularly difficulties due to concurrency. The primary mechanism for avoiding deadlock is the style that a pull should return `null` if no data are available. The danger of livelock is largely eliminated by avoiding feedback loops, although several interesting models include feedback loops that do not livelock because of the logic contained in components (see [31] section 2.6). Data races do not occur accidentally because methods execute atomically. Nonetheless, on a coarser level, nondeterministic interactions like those in the sensor fusion example are easy to define. Indeed, these kinds of interactions are common in the application domain that Click targets, network routers.

3.3 Others

There are many other models with imperative concurrent semantics. Here we briefly mention some that have been applied to the design of embedded systems.

3.3.1 Bluespec

In a Bluespec [2, 43] model, components not only contain methods, but also *activation rules* and *execution constraints*. Each activation rule describes an atomic state update in the system, which can be performed whenever the associated execution constraints are satisfied. Bindings between methods enable complex state updates to be specified compositionally as a group of methods.

Conceptually, state updates in a Bluespec system occur sequentially. However, in some cases activation rules operate on independent portions of the system state, in which case they are called *conflict free*. These *conflict-free* rules represent parallelism in a system and can be executed concurrently. Bluespec discovers conflict free rules through static program analysis and generates run-time scheduling logic for selecting sets of rules which

Livelock in Bluespec models is prevented by a requirement that no method can cause itself to be executed through a sequence of method invocations. This requirement is guaranteed through static analysis of component compositions. Deadlock in Bluespec models cannot generally be avoided, since it is possible that a state in execution is reached where there are no activation rules whose execution constraints can be satisfied.

Bluespec has seen significant application in the specification of digital logic circuits [25, 26]. Current compilers map a composition to a synchronous circuit that executes an activation rule in a single cycle. Methods are converted into combinational logic, which is guaranteed to be acyclic given the constraints on re-entrant methods. In each cycle every rule executes concurrently, but the results are gated so that only the state updates corresponding to a set of conflict-free rules are committed to the system state.

Bluespec models can also be synthesized directly into sequential software, which can be used to efficiently simulate synthesized digital logic systems. In software, it is more efficient to make scheduling decisions for activation rules initially and to only execute code corresponding to a single activation rule at a time. In comparison with direct simulation of synthesized digital logic, this technique offers significant speedup for many applications, since only committed activation rules are actually executed. Additionally, given coarse-grained activation rules, it seems possible to execute more than one rule in software concurrently.

3.3.2 Koala

Koala [49, 48] is a model and language for components with procedure-call interfaces and a visual syntax with "provides" and "requires" ports that get connected. It has been proposed for use in the design of consumer electronics software specifically. As with nesC/TinyOS and Click, in a Koala model, connections between components represent method bindings. Communication occurs through method arguments and return values and the interaction between communicating components is primarily sequential. Koala allows components to contain arbitrary code and perhaps to encapsulate arbitrary operating system threads.

A "configuration" is an interconnection of components plus configuration-specific code in something called a "module." To get hierarchy, the configuration can export

its own requires and provides interfaces, and these can be mediated by the module. E.g., the module can translate a particular provided method into a sequence of calls to provided methods of the components (e.g. to initialize all the components). The module is configuration specific, and is not itself a component, so it does not pollute the component library. The module can also provide services that are required by the components. For example, a component may require values for configuration parameters, and the module can provide those values. Partial evaluation is used to avoid introducing overhead in doing things this way.

Modules offer a much richer form of hierarchical abstraction than either nesC or Click provide. Modules are also used to implement primitive components, thus providing the leaf cells of the hierarchy.

Each "requires" interface must be connected to either a module or a "provides" interface (input port). A "provides" interface, however, can be connected to zero or more "requires" interfaces. An example is given in [49] where components require a particular hardware interface (an I2C bus) that must be provided by a configuration. Operating system and scheduling services also interact with components through requires and provides interfaces. Thus, the language provides clean mechanisms for relating hardware requirements to software services.

A limited form of dynamic binding is provided in the form of "switches," which work together with a module to direct procedure calls. These can be used at run time to direct a method call to one or another component. Switches can also be used with "diversity interfaces" (see below), in which case, partial evaluation will likely lead to static binding and the elimination of some components from a configuration (components that are not used).

"Diversity" means one definition, multiple products. Koala's features support this well, particularly through its partial evaluation and static binding, which avoid the overhead often incurred by making components flexible. The authors compare the use of "requires" interfaces to property lists in more conventional component architectures with set() and get() methods, and point out that set() and get() make it more difficult to optimize when properties are set at design time. Instead of "providing" interfaces that must be filled in by the configuration (e.g. set()), Koala components have "required" interfaces that the configuration must provide. These are called "diversity interfaces."

Koala components can provide "optional interfaces" (fashioned after COM's query interface mechanism), which are automatically extended with an isPresent function, which the component is required to implement. E.g., the presence of an interface may depend on the hardware configuration. A component may also require an "optional interface" (which is, to be sure, odd terminology), in which case the component can query for whether a configuration has a matching "provides" interface.

The hierarchical structure, components with provides and requires interfaces, and bindings concepts come from the architecture description language Darwin [41], but the modules and diversity schemes are new.

4 Declarative Concurrent Models

As mentioned above, Simulink and SCADE 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. In this section, we examine a few of the models of computation with this character.

4.1 Simulink

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 6 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 6 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.

The discrete controller shown in figure 6 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, as discussed below). 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 7. 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 [38].

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 6. This is because the execution of the slow block will typically stretch over several executions of the fast block, as shown in figure 7.[7] 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 [23], 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

[7] This schedule is simplified, showing only the invocations of the methods associated with the blocks that produce outputs.

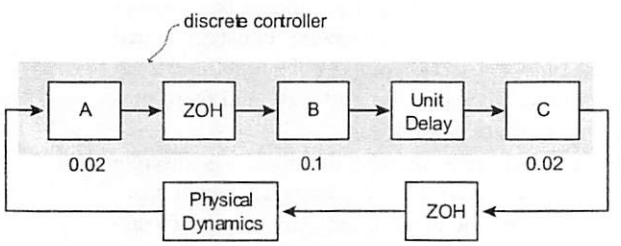

Figure 6. A representation of a Simulink program.

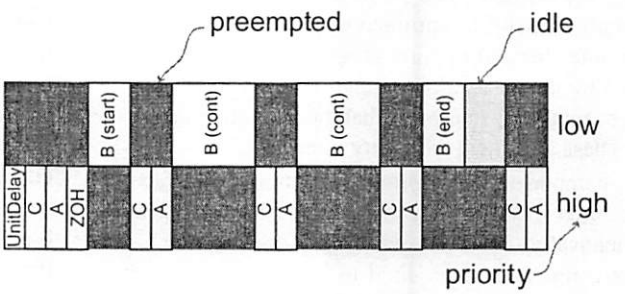

Figure 7. A simplified representation of a Simulink schedule.

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 does have some weaknesses, however. The sensor fusion problem that we posed earlier does not match its discrete multitasking model very well. While it would be straightfoward to construct a discrete multitasking model that polls the sensors are 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.

4.2 Discrete-Event

In figure 2, we gave 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), in computer network modeling and simulation (OPNET Modeler[8] and Ns-2[9], for example), and in 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 chronological 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 [13] or the speculative execution methods of Jefferson [28]. 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 [39, 19]. A significant challenge is to achieve the timed semantics efficiently while building on software abstractions that have abstracted away time.

4.3 Synchronous Languages

SCADE [4], a commercial product of Esterel Technologies, builds on the synchronous language Lustre [22]. Of the flagship synchronous languages, Esterel [5], Signal [21], and Lustre, Lustre is the simplest in many respects. All the synchronous languages have strong formal properties that yield quite effectively to formal verification techniques, but the simplicity of Lustre in large part accounts for SCADE achieving a widely covetted certification for use in safety critical embedded avionics software by European agencies.

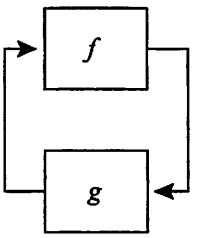

Figure 8. A simple feedback system illustrating the fixed point principles of synchronous languages.

The principle behind synchronous languages is simple, although the consequences are profound [3]. Execution follows "ticks" of a global "clock." At each tick, each variable (represented visually by the wires that connect the blocks) may have a value (it can also be absent, having no value). Its value (or absence of value) is defined by functions associated with each block. That is, each block is a function from input values to output values. In figure 8, the variables x and y at a particular tick are related by

$$x = f(y), \text{ and}$$

$$y = g(x).$$

The task of the compiler is to synthesize a program that, at each tick, solves these equations. Perhaps somewhat surprisingly, this turns out to be not difficult, well-founded, and reasonably efficient.

An interesting issue with Lustre is that it supports multiple rates. That is, the master clock can be "divided down" so that certain operations are performed on only some ticks of the clock. There is a well-developed formal "clock calculus" that is used by the compiler to analyze systems with such multirate behavior. Inconsistencies are detected by the compiler.

In SCADE, the functions associated with blocks can be defined using state machines. They can have behavior that changes with each tick of the clock. This offers an expressive and semantically rich way to define systems, but most interestingly, it also offers opportunities for formal verification of dynamic behavior. As long as the state machines have a finite number of states, then in principle, automated

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

tools can explore the reachable state space to determine whether safety conditions can be guaranteed.

The nondeterministic merge of figure 1 is not directly supported by Lustre. The synchronous language Signal [21] extends the principles of Lustre with a "default" operator that supports such nondeterministic merge operations. The timed behavior of figure 2 is also not directly supported by Lustre, which does not associate any metric with the time between ticks. Without such a metric, the merging of sensor inputs in figure 2 cannot be done deterministically. However, if these events are externally merged (for example in the interrupt service routines, which need to implement the appropriate mutual exclusion logic), then Lustre is capable of expressing the rest of the processing. The fact that there is no metric associated with the time between ticks means that Lustre programs can be designed to simply react to events, whenever they occur. This contrasts with Simulink, which has temporal semantics. Unlike Simulink, however, Lustre has no mechanisms for multitasking, and hence long running tasks will interfere with reactivity. A great deal of research has been done in recent years in "desynchronizing" synchronous languages, so we can expect in the future progress in this direction.

4.4 Dataflow

As with the other models of computation considered here, components in a dataflow model of computation also encapsulate internal state. However, instead of interacting through method calls, continuous-time signals, or synchronously defined variables, components interact through the asynchronous passing of data messages. Each message is called a *token*. In this section, we will deal only with models where messages are guaranteed to be delivered in order and not lost. For these models it is common to interpret the sequence of tokens communicated from one port to another as a (possibly infinite) *stream*. It is not uncommon to use visual representations for dataflow systems, as in figure 9. In that figure, the wires represent streams, the blocks represent dataflow *actors*, and the triangles represent *ports*. Input ports point into the block, and output ports point out. Feedback is supported by most variants of dataflow semantics, although when there is feedback, there is risk of deadlock.

There are many variants of dataflow semantics. We consider a few of them here.

4.4.1 Kahn Process Networks

Figure 1, discussed above, has the semantics of Kahn process networks (PN) [30, 37] augmented with a nondeterministic merge [1, 15]. In PN semantics, each actor executes (possibly forever) in its own thread of control. The connections between actors represent streams of tokens. In

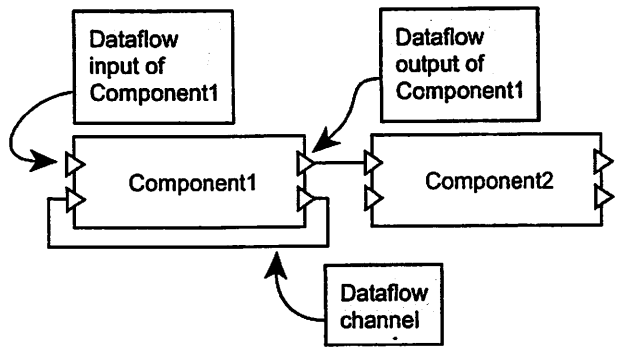

Figure 9. A diagram representing dataflow-oriented components.

Kahn/MacQueen semantics [30], the way that threads interact with the ports has a key constraint that guarantees determinacy. Specifically, a thread is not permitted to "ask" an input port whether there are available tokens to read. It must simply read from the port, and if no tokens are available, the thread blocks until tokens become available. This behavior is called "blocking reads." Correspondingly, when the thread produces an output token, it simply sends it to the output port and continues. It is not permitted to ask the output port whether there is room for the token, or whether the ultimate recipient of the token is ready to receive it. These simple rules turn out to sufficient to ensure that actors implement *monotonic* functions over streams, which in turn guarantees determinacy [29]. Determinacy in this case means that every execution of the PN system yields the same stream on tokens on each connection. That is, the PN system *determines* the streams.

In figure 1, 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. This behavior is not monotonic. In fact, it cannot be implemented with blocking reads in a single actor thread. Extensions of PN that support such nondeterministic operations turn out to be especially useful for embedded software, and have been an active area of research [1, 15].

A key issue with PN models is that they may deadlock. They may also consume unbounded memory buffering tokens between actors. It turns out that it is undecidable whether a PN model deadlocks or executes in bounded memory. This means that no algorithm exists that can always answer these questions in finite time. Nonetheless, there are simple execution policies that guarantee that *if* a particular PN system *can* be executed without deadlock in bounded memory, then it will be executed without deadlock in bounded memory [46]. The undecidable problem is

solved by a runtime policy, which does not need to solve the problem in bounded time. A practical implementation this policy is available in the Ptolemy II system [14].

4.4.2 Dennis Dataflow

In a distinct family of dataflow models of computation, instead of executing a (possibly infinite) thread, a component executes a sequence of distinct *firings*. This style of dataflow model was introduced by Dennis in the 1970s [16], and was applied to the design of high performance computer architectures for several years. Semantically, the sequence of firings, of course, can be considered to be a thread with a limited mechanism for storing state, so at a fundamental level, the distinction between PN and Dennis dataflow is not great [37]. But it turns out to be particularly convenient to formulate dataflow systems in terms of firings. A great deal of formal analysis of the system is enabled by this abstraction.

A firing is enabled by satisfaction of a *firing rule*. The formal structure of firing rules has considerable bearing on the formal properties of the model as a whole [34]. Each firing reads a short sequence of input tokens and produces a short sequence of output tokens. The firing of a dataflow component might also update the internal state of a component, affecting the behavior of the component in future firings.

There are two common ways of implementing dataflow models. One possibility is to implement a centralized runtime scheduler that selects and executes actors whose firing rules are satisfied. A second possibility is to statically analyze the dataflow graph and construct a static, finite description of the schedule. The latter approach is preferable for embedded software, since the static analysis also yields execution time and memory usage information. However, for general dataflow models, it turns out to be undecidable whether such static schedules can be constructed [8]. A suite of decidable special cases of dataflow have been developed over the years, however, and some of these are quite promising for embedded software systems.

4.4.3 Decidable Dataflow Models

A simple special case of dataflow models restricts actors so that on each port, they produce and consume a fixed, prespecified number of tokens. This model of computation has been called *synchronous dataflow* (SDF) [36], but to avoid confusion with the (significantly different) synchronous languages (see [22] for example), it would perhaps better be called *statically schedulable dataflow* (SSDF). Indeed, the key feature of this model of computation is that simple static analysis either yields a static schedule that is free of deadlock and consumes bounded memory, or proves that no such schedule exists [36].

Because of the constraint that actors produce and consume only fixed, pre-specified numbers of tokens on each firing, SSDF by itself cannot easily describe applications with data-dependent control structure. A number of extensions enrich the semantics in various ways.

Boolean dataflow[8][11] (BDF) and integer-controlled dataflow[9] (IDF) augment the model by permitting the number of tokens produced or consumed at a port to be symbolically represented by a variable. The value of this variable is permitted to change during execution, so data-dependent control flow can be represented. Static analysis can often still be performed, but in principle, it is undecidable whether a BDF or IDF program can execute without deadlock in bounded memory. Nonetheless, for many practical programs, static analysis often yields a proof that it can, and in the process also yields a *quasi-static schedule*, which is a finite representation of schedule with data-dependent control flow.

The fact that BDF and IDF are undecidable formalisms, however, is inconvenient. Static analysis can fail to find a schedule even when such a schedule exists. Cyclo-static dataflow (CSDF) [7] offers slightly more expressiveness than SDF by permitting the production and consumption rates at ports to vary periodically. SDF can also be combined hierarchically with finite state machines (FSMs), and if the state transitions are constrained to occur only at certain disciplined times, the model remains decidable. This combination has been called heterochronous dataflow (HDF) [20]. Parameterized SDF [6] offers similarly expressive variability of production and consumption rates while remaining within a decidable formalism. Most of these variants of dataflow are available in the Ptolemy II system [14] or in Ptolemy Classic [10].

4.5 PECOS

A final model that we consider shares a number of features with the previous, but also has some unique properties. In a PECOS [50, 18, 42] model, there are three types of components: *active*, *event*, and *passive*. These components are composed hierarchically with the constraint that an active component must occur at the root of the tree. Active components are associated with an independent thread that is periodically activated. Event components are similar to active components, except they are triggered by aperiodic events occurring in the system. Event components are generally associated with sensors and actuators in the system and are triggered when a sensor has data or an actuator requires data. Passive components are executed by the element that contains them.

Connections between represent a variable in shared memory that is read and written by the components connecting to it. Each passive component is specified by a

single `execute()` method that reads the appropriate input variables and writes the correct output variables. The simplest PECOS model consists of an active component at the toplevel, containing only passive components. The entire execution occurs in the single thread, and consists of sequenced invocations of the `execute()` methods.

Active and event components are specified by an `synchronize()` method, in addition to the `execute()` method. In order to avoid data races, variables for communicating with active and event components are double buffered. The `synchronize()` method is executed by the component's container to copy the input and output variables. The `execute()` method that actually performs processing only access the variable copies.

5 Conclusion

The diversity and richness of semantic models for embedded software is impressive. This is clearly a lively area of research and experimentation, with many innovative ideas. It is striking that none of the concurrent models of computation considered in this paper rely on threads as the principle concurrency mechanism. Yet prevailing industrial practice in embedded software often does, building the software by creating concurrent threads and using the mutual exclusion and semaphore mechanisms of a real-time operating system to manage concurrency issues. We argue that these mechanism are too difficult for designers to understand, and that except in very simple systems, should not be used in raw form. At a minimum, a design pattern corresponding to a clean concurrent model of computation (such as process networks or synchronous component composition) is required to achieve truly reliable systems. But better than informal use of such design patterns is the use of languages or frameworks that enforce the pattern and provide proven implementations of the low-level details. We have outlined the key features of a few such promising languages and frameworks.

References

[1] Arvind and J. D. Brock. Resource managers in functional programming. *J. of Parallel and Distributed Computing*, pages 5–21, 1984.

[2] L. Augustsson, J. Schwartz, and R. Nikhil. Bluespec language definition. Technical report, Sandburst Corporation, Nov. 2000.

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

[4] G. Berry. The effectiveness of synchronous languages for the development of safety-critical systems. White paper, Esterel Technologies, 2003.

[5] G. Berry and G. Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[6] B. Bhattacharya and S. S. Bhattacharyya. Parameterized dataflow modeling of dsp systems. In *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, Istanbul, Turkey, 2000.

[7] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete. Static scheduling of multi-rate and cyclo-static dsp applications. In *Workshop on VLSI Signal Processing*. IEEE Press, 1994.

[8] J. T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*. Ph.d. thesis, University of California, Berkeley, 1993.

[9] J. T. Buck. Static scheduling and code generation from dynamic dataflow graphs with integer-valued control systems. In *IEEE Asilomar Conf. on Signals, Systems, and Computers*, Pacific Grove, CA, 1994.

[10] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *Int. Journal of Computer Simulation, special issue on "Simulation Software Development"*, 4:155–182, 1994.

[11] J. T. Buck and E. A. Lee. Scheduling dynamic dataflow graphs with bounded memory using the token flow model. In *IEEE Int. Conf. on Acoustics, Speech, and Signal Processing (ICASSP)*, volume 1, pages 429–432, 1993.

[12] K. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM*, 24(11):198–206, April 1981.

[13] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison Wesley, 1988.

[14] J. Davis et al. Ptolemy II - Heterogeneous concurrent modeling and design in Java. Memo M01/12, UCB/ERL, EECS UC Berkeley, CA 94720, Mar. 2001.

[15] E. de Kock, G. Essink, W. Smits, P. van der Wolf, J. Brunel, W. Kruijtzer, P. Lieverse, and K. Vissers. Yapi: Application modeling for signal processing systems. In *Proceedings of the 37th Design Automation Conference (DAC'2000)*, pages 402–405, June 2000.

[16] J. B. Dennis. First version of a dataflow procedure language. In *Programming Symposium: Proceedings, Colloque sur la Programmation*, number 19 in Lecture Notes in Computer Science, pages 362–376. Springer, Apr. 1974.

[17] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of Programming Language Design and Implementation (PLDI)*, June 2003.

[18] T. Genßler, A. Christoph, M. Winter, O. Nierstrasz, S. Ducasse, R. Wuyts, G. Arévalo, B. Schönhage, P. Müller, and C. Stich. Components for embedded software: the PECOS approach. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 19–26, 2002.

[19] 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.

[20] 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), 1999.

[21] P. L. Guernic, T. Gauthier, M. L. Borgne, and C. L. Maire. Programming real-time applications with signal. *Proceedings of the IEEE*, 79(9), 1991.

[22] 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.

[23] 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.

[24] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for network sensors. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Nov. 2000.

[25] J. C. Hoe and Arvind. Hardware synthesis from term rewriting systems. In *International Conference on Very Large Scale Integration (VLSI)*, IFIP Conference Proceedings, pages 595–619. Kluwer, Dec. 1999.

[26] J. C. Hoe and Arvind. Synthesis of operation-centric hardware descriptions. In *Proceedings of International Conference on Computer Aided Design (ICCAD)*, Nov. 2000.

[27] P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3), 1989.

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

[29] G. Kahn. The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress 74*. North-Holland Publishing Co., 1974.

[30] G. Kahn and D. B. MacQueen. Coroutines and networks of parallel processes. In B. Gilchrist, editor, *Information Processing*. North-Holland Publishing Co., 1977.

[31] E. Kohler. *The Click modular router*. PhD thesis, Massachusetts Institute of Technology, Nov. 2000.

[32] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3), Aug. 2000.

[33] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, Reading MA, 1997.

[34] E. A. Lee. A denotational semantics for dataflow with firing. Technical Report UCB/ERL M97/3, Electronics Research Laboratory, Berkeley, CA 94720, 1997.

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

[36] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 1987.

[37] E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, 1995.

[38] 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.

[39] J. Liu and E. A. Lee. Timed multitasking for real-time embedded software. *IEEE Control Systems Magazine*, pages 65–75, 2003.

[40] S. Maffeis. Adding group communication and fault-tolerance to CORBA. In *Proceedings of the USENIX Conference on Object-Oriented Technologies (COOTS)*, pages 16–29, June 1995.

[41] J. Magee and e. al. Specifying distributed software architectures. In *ESEC '95*, pages 137–153, Berlin, 1995. Springer-Verlag.

[42] O. Nierstrasz, G. Arévalo, S. Ducasse, R. Wuyts, A. Black, P. Müller, C. Zeidler, T. Genßler, and R. van den Born. A component model for field devices. In *Proceedings International Working Conference on Component Deployment*. IFIP/ACM, June 2002.

[43] R. Nikhil. Bluespec SystemVerilog: Efficient, correct RTL from high-level specifications. In *Conference on Methods and Models for Codesign (MEMOCODE)*, June 2004.

[44] Object Management Group (OMG). CORBA event service specification, version 1.1, March 2001.

[45] P. Panangaden and V. Shanbhogue. The expressive power of indeterminate dataflow primitives. *Information and Computation*, 98(1), 1992.

[46] T. M. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, EECS Department, University of California at Berkeley, CA, 1995.

[47] N. Shah, W. Plishker, and K. Keutzer. Np-click: A programming model for the intel ixp1200. In P. Crowley, M. Franklin, H. Hadimioglu, and P. Onufryk, editors, *Network Processor Design: Issues and Practices, Volume 2*, pages 181–201. Elsevier, 2004.

[48] R. van Ommering. Building product populations with software components. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 255–265, May 2002.

[49] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala component model for consumer electronics software. *IEEE Computer*, 33(3):78–85, Mar. 2000.

[50] M. Winter, T. Genßler, A. Christoph, O. Nierstrasz, S. Ducasse, R. Wuyts, G. Arévalo, P. Müller, C. Stic, and B. Schönhage. Components for embedded software : The PECOS approach. In *Workshop on Composition Languages, In conjunction with 16th European Conference on Object-Oriented Programming (ECOOP)*, June 2002.