Timed Multitasking for Real-Time Embedded Software

©JENE NELSON AT CN GRAPHICS INC.

By Jie Liu and Edward A. Lee

Software in real-time embedded systems differs fundamentally from its desktop or Internet counterparts. Embedded computing is not simply computation on small devices. In most control applications, for example, embedded software engages the physical world. It reacts to physical and user-interaction events, performs computation on limited and competing resources, and produces results that further impact the environment. Of necessity, it acquires some properties of the physical world, most particularly, time.

Despite the fact that both *value* and *time* affect the physical outputs of embedded systems, these two aspects are developed separately in typical embedded software design. The functionality is determined at design time with assumptions such as zero or a fixed nonzero run-time delay. The actual timing properties are determined at run time by a real-time operating system (RTOS). Typically, an RTOS offers as control of these timing properties one number for each task, a *priority*. Whether a piece of computation can be finished or brought to a quiescent state at a particular time is totally a dynamic phenomenon, and it depends largely on the hardware platform, when the inputs arrive, what other software is running at that time, and the relative priorities. These factors are usually out of the control of embedded system designers and may break the timing assumptions that the control algorithms may rely on. In most control applications, this run-time uncertainty is undesirable or even disastrous.

We believe that two steps can be taken to improve the design process for embedded software and to bridge the gap between the functionality development and timing assurance:

- rigorous software architectures that expose resource utilization and concurrent interactions among software components
- specification, compilation, and execution mechanisms that preserve timing properties throughout the software life cycle.

A component-based software architecture can help compilers to determine the logical dependencies and shared resources among components. By bringing the notion of time and concurrent interaction to the programming level, compilers and run-time systems can be developed to preserve both timing and functional properties at run time. Recent innovations in real-time programming models such as port-based objects (PBOs) [1] and Giotto [2] are examples that take a time-triggered approach to scheduling software components and to preserving their timing properties. These purely time-triggered approaches, although explicitly controlling the timing of each component, require tasks to be periodic and do not handle irregularly spaced new information (or events) well.

Liu (jieliu@parc.com) is with Palo Alto Research Center, 3333 Coyote Hill Rd., Palo Alto, CA 94304, U.S.A. Lee is with the Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720, U.S.A.

1053-5888/03/$17.00©2003IEEE

In this article, we introduce an event-triggered programming model, timed multitasking (TM), that also takes a time-centric approach to real-time programming but controls timing properties through deadlines and events rather than time triggers. By doing so, each piece of information is processed exactly once, and the tasks can be aperiodic. This model takes advantage of an actor-oriented software architecture [3] and embraces timing properties at design time, so that designers can specify when the computational results are produced to the physical world or to other actors. The specification is then compiled into stylized real-time tasks, and a run-time system further ensures the function and timing determinism during execution. As long as there are sufficient resources, the computation will always produce predictable values at a predictable time.

Real-Time Programming: Common Practice

Real-time systems typically need to perform multiple tasks at the same time. Each invocation of a task is a finite amount of computation that requires some resources and takes some time to perform. Tasks may compete for resources, such as CPU, I/O access, or network bandwidth; thus, a resource manager is needed to allocate resources and schedule task activation. This resource management is a major responsibility of real-time operating systems in common embedded systems. When two eligible tasks are competing for resources, the operating system must choose to grant the resources to one of them, and, as a consequence, that task finishes sooner.

The process of choosing which task to grant resources to is called real-time scheduling. A typical strategy is to assign priorities to tasks and to execute the highest priority task that is eligible. Intuitively, priorities represent the relative importance of tasks at run time. Priorities can be statically assigned to tasks at design time, or they may be dynamically determined at run time. In today's embedded systems, given the run-time overhead of computing priorities, it is common to fulfill timing constraints by statically assigning priorities among the tasks.

Another powerful concept for dealing with timing properties is the notion of preemption, which is to pause a running task, say A, and execute another task, say B. By doing so, task B, although it is activated later than task A, can finish before task A finishes. Obviously, task B must have a higher priority to preempt task A.

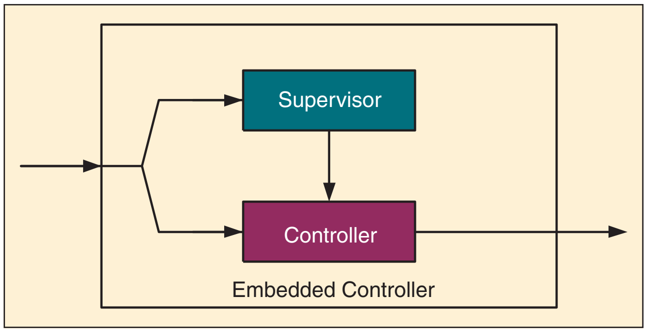


Figure 1. *An embedded controller with two tasks.*

Consider the example shown in Figure 1, where a Controller and a Supervisor are implemented on the same computer. Suppose that the controller is triggered by periodic samples, say, every 2 ms, and for each sampling input, the controller produces an output with a delay. For now, let's assume the delay is fixed, say, 1 ms, from activation to activation. The supervisor task is only triggered once in a while (e.g., every second), and it takes a long time (> > 2 ms) to compute a new set of parameters to adjust the control algorithm. Since both tasks are implemented on the same embedded system, they share the computing resources. Assume it is unacceptable for the controller to stop producing any output for a long time if the supervisor task keeps running. A preemptive strategy is shown in Figure 2, where the controller task preempts the long-run supervisor task. In this figure, each box represents the execution of a task. The nonshadowed parts within the execution of the supervisor task indicate that its execution is preempted by the controller task.

Real-Time Scheduling

How to assign priorities to multiple tasks is a key part of the real-time scheduling problem. This has been an active research area for more than 20 years, starting with the seminal work by Liu and Layland [4]. The goal of real-time scheduling is to devise a set of priority assignment rules to make sure that all tasks are finished before their deadlines.

Real-time scheduling algorithms typically make some assumptions about tasks and resources. For example, Liu and Layland's original work makes the following assumptions:

- a single and arbitrarily preemptable resource (CPU)
- independent tasks
- fixed and known task execution times
- each task must be completed before receiving the next trigger
- periodic task triggers (for rate monotonic scheduling).

Under these assumptions, Liu and Layland derived rate-monotonic (RM) and earliest-deadline-first (EDF) scheduling policies. These algorithms are shown to be optimal (in terms of CPU utilization) for static and dynamic priority assignments, respectively. Further work in this area has developed more sophisticated timing analysis theories and has relaxed many assumptions in the original algorithms [5]-[9]. However, most of them still rely on knowing all tasks' worst-case execution time (WCET) and having arbitrary preemptability. When multiple resources are being managed, optimal scheduling becomes NP-hard [10], so most methods deal with only one shared resource.

In reality, many of the assumptions in scheduling theories do not hold. Tasks may require multiple resources to execute, and they can be strongly coupled. For instance, in our controller/supervisor example, the supervisor may update the controller's parameters by directly accessing them. Suppose that the lower-priority supervisor task is writing to a set of parameters in the controller, and at the same time, the higher-priority controller task is activated. The controller task cannot start immediately since its parameters may not be consistent. To ensure the integrity of the parameters, the controller can only preempt the supervisor after the update operations are finished. We call this situation partial preemptability. Partial preemptability not only complicates timing analysis in the sense that the response time of the controller is lengthened, but it may also cause more serious problems, such as priority inversion.

Priority Inversion

The intuition behind assigning priorities to tasks is to prioritize resource utilization and obtain fast response for critical tasks. However, because of the partial preemptability of some tasks, blindly following the priority assignment and triggering high-priority tasks may cause a high-priority task to be blocked by low-priority tasks indefinitely, a situation called *priority inversion*.

Consider the following situation in our previous example, where there is one more task, say, fault diagnosis, that is independent of the controller and the supervisor, running on the same embedded computer. Suppose that the fault diagnosis task has an intermediate priority that is higher than that of the supervisor but lower than that of the controller. Suppose in addition that at some point, when the controller is blocked waiting for the supervisor to finish updating the control parameters, the fault diagnosis task is activated. Since the fault diagnosis task is independent of the supervisor and has a higher priority, it preempts the supervisor and starts executing. Now the controller, although having a higher priority than the fault diagnosis task, is in fact blocked by the fault diagnosis task. This situation is illustrated in Figure 3. Imagine that there are multiple intermediate-priority tasks that act like the fault diagnosis task here. Then the controller task may be postponed indefinitely.

Priority inversion problems are usually solved by the priority inheritance and priority ceiling protocols [11]. The basic idea of these protocols is to look into the content of each task, analyze critical sections (e.g., shared data access), and for each critical section, find the highest priority task that may access it. Call this highest priority value p. Then, if a lower priority task A enters this section, the priority of task A becomes p, so that no task of priority lower than p can preempt A. When the task leaves the critical section, its priority drops back to its normal value.

Priority inheritance and priority ceiling protocols successfully solve the priority inversion problem. By adding more constraints, such as requiring that all tasks enter critical sections in the same order, it also solves deadlock problems that may be caused by cross waiting on data access. Thus, these protocols are widely implemented in real-time operating systems like VxWorks, QNX [12], and Resource Kernel [13]. On the down side, the footprint and the run-time overhead of these protocols are not trivial. Thus, some lightweight real-time kernels do not support them and require software designers to take care of avoiding priority inversion and deadlock problems themselves.

More Pitfalls

Using priorities as the only tuning parameter and applying priority-driven preemptive execution rules without considering the status of other tasks introduce many problems. Besides priority inversion, other pitfalls exist when the assumptions of real-time scheduling theories do not hold.

- Preemptive execution, especially with static priority assignment, is fundamentally fragile. The timing behavior of tasks may be very sensitive to task activation time and their actual execution time. An early arrival of a high-priority task or an unexpectedly long execution of a task can have domino effects and add delays to the response time of all subsequent lower-priority tasks.

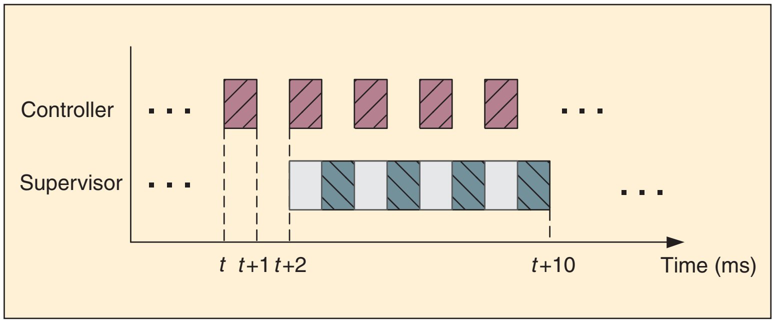


Figure 2. *Preemptive execution of two tasks.*

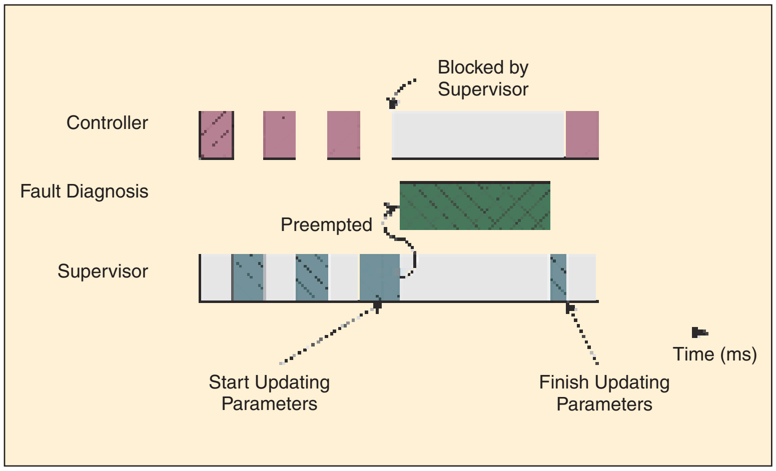


Figure 3. *Illustration of priority inversion. A high-priority controller task is blocked by the mid-priority fault diagnosis task, since the controller is blocked by the low-priority supervisor due to data dependency.*

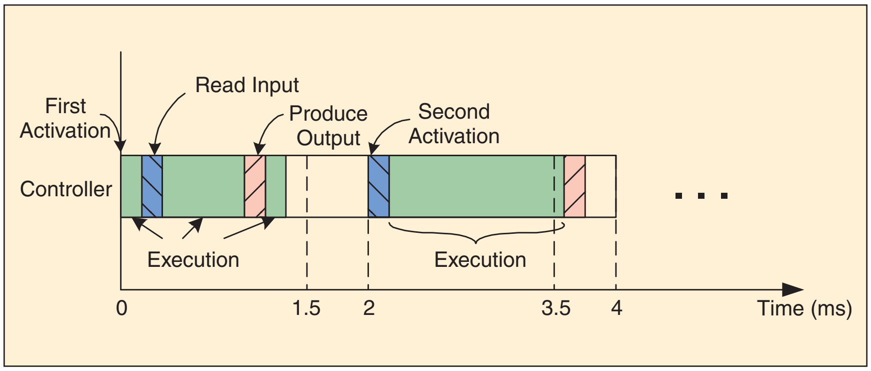


Figure 4. *In PBO models, tasks are activated by time, and they read from and write to a global data space.*

- The results of schedulability analysis may not be very useful. The typical answers from a schedulability analysis are the worst-case response time between the triggering and the finishing of the tasks. These values are required to be less than the deadlines to pass the schedulability test. However, schedulability analysis does not reveal how often the worst-case response time is met, what distribution it has, and what happens if it is greater than the deadline. In many control applications, the physical system is fundamentally robust, and missing a deadline occasionally may not cause catastrophic results.
- The WCET may not be the best representation of the execution time of a task. It could be much larger than the average execution time, and by using WCET for schedulability analysis, the results could be very conservative. As a consequence, the resources are not sufficiently used to be cost effective.
- Pushing for fast response may not be optimal. Some hard real-time algorithms may have strict requirements on the output time. An optimal result may only be achieved by emitting the output at a particular time. Early as well as late outputs may result in a suboptimal result. This is particularly true for some multimedia applications and predictive control algorithms.

One fundamental problem behind these pitfalls is that there is not enough discipline in real-time programming. Common embedded software development inherits empirical programming models from desktop software development and tries to patch timing properties by tuning priorities after fixing the functionality. Characteristics such as resource requirements, synchronization, and critical sections, which directly affect timing properties, are not explicit parts of the program specification. Thus, there is not much space for compilers and run-time systems to help ensure timing properties at run time. This in turn forces designers to work with WCET and worry about the domino effects and to rely on exhaustive testing to develop confidence in a design.

Time-Triggered Architectures

Time-triggered architectures (TTAs) are system architectures and network protocols for safety-critical distributed embedded systems [14]. Various programming models use the same ideas to address the fundamental issues of real-time programming by using time as the only trigger for computation. They bring timing constraints explicitly to the programming model level, so that compilers and run-time systems can schedule and optimize the software to achieve timing determinism. We give two examples of time-triggered models in this section, port-based objects and Giotto, that significantly influence our design of the timed multitasking model.

Port-Based Objects

In PBO models [1], tasks, called port-based objects, are time triggered, and the communication channels between them form a global data space. Once activated, a PBO is free to read from and write to the global data space. Read and write operations are atomic, and all computation within the PBOs is based on their internal variables. Although synchronization, using monitors and locks, for example, is still necessary to guard the access to the global data space, it is managed outside of the objects, at the operating system level, instead of by the objects themselves. It is thus much easier to maintain the atomicity of the communications and avoid priority inversion problems.

The execution time of a PBO task may vary from activation to activation, and the time when the inputs are consumed and the outputs are produced may not be regular. The global data space has a state semantics, meaning that the value of a variable is preserved as long as it is not overwritten, and a newly written value will overwrite the old value regardless of whether the old one has been used by other tasks. An example of an execution trace of a Controller PBO is shown in Figure 4, where the controller is activated every 2 ms. Suppose that a successor task reads the controller's output at 1.5 and 3.5 ms in this timing diagram; then in the first period, it reads a fresh output from the controller, but in the second period, since a new output is not yet produced, it will use the controller output from the last cycle again.

Giotto

The Giotto model [2] further extends the time-triggered concepts so that both computation and communication among tasks are triggered by time. Each execution of a task has a well-defined start and stop time, declared at design time. Conceptually, communication only occurs between task executions. A task obtains all inputs at the start time of its execution, and the outputs are only available to the rest of the system at the stop time of the execution, regardless of when the values of outputs are actually calculated.

For example, suppose we specify that a controller is activated every 2 ms. In Giotto, this implies that the input data is consumed at the beginning of the 2 ms, the execution happens somewhere within one period, and the output is made available at the end of the period. A timing diagram is shown in Figure 5. Thus, no matter how quickly or slowly the controller executes, as long as it can finish in 2 ms, the outputs will be produced at the exact time instants.

Tasks in Giotto execute concurrently, at least conceptually. Consider, for instance, a Giotto model as shown in Figure 6, where a task A with period 2 ms feeds data to task B with period 2 ms. Conceptually, the tasks begin executing simultaneously and end simultaneously. Thus, task B will always see data provided by A in the previous cycle. Hence there is a precise 2-ms delay introduced by task A. This one sample delay per task may not be desirable for some applications, but it yields a deterministic timing and functional behavior.

Notice that both PBO and Giotto models use the state semantics for communication, which has the significant advantage of avoiding unnecessary synchronizations among senders and receivers and allowing tasks to be triggered at any time. However, state semantics may lose the notion of new events such that some data may be processed more than once and some others may be completely ignored. The problem is left to the designers in the PBO model, whereas it is made explicit by the one-sample delay in the Giotto model. Furthermore, pure time-triggered models require precise time synchronizations across distributed platforms, which may be a significant cost for complex systems.

Timed Multitasking

This article describes the TM model, which tackles the real-time programming problem using an event-driven approach. Unlike in the PBO and Giotto models, a task in TM (called an actor) is executed when there are input events that fulfill certain conditions specified by the actor. But like the Giotto model, it provides a predictable input/output timing so that the computational delays can be used as deterministic parameters in algorithm designs. The basic idea behind the TM model is simple: since the activation of a task depends either on other tasks or on interrupts, by controlling the time at which outputs are produced and triggering tasks with new events, we can effectively control both the starting and stopping time of each task, thus obtaining deterministic timing properties.

Actor-Oriented Programming

The TM model is built on top of a component-based approach, where components are called Ptolemy actors (the name is due to its implementation in the Ptolemy project [15]). This model helps the designer to isolate atomic computations and identify interdependencies among software components. This notion of actors differs from Agha's actor model in the sense that actors do not necessarily associate with their own thread of control.

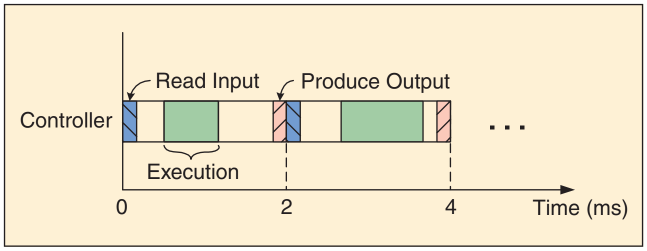


Figure 5. *In Giotto, regardless of the exact execution time of a task, the inputs are consumed and the outputs are produced at well-defined time instances.*

In TM, an actor represents a sequence of reactions, where a reaction is a finite piece of computation. The actor has state, which carries from one reaction to another. Actors have ports, which are their communication interface. The concept of ports is broad enough to support many communication mechanisms among actors. For example, a port may represent an interrupt, a first-in-first-out (FIFO) queue, a rendezvous point, etc. Actors can only communicate with other actors and the physical world through ports. Thus, their internal state is not directly accessible by anything outside the actor.

Unlike method calls in object-oriented models, interaction with the ports of an actor may not directly transfer the flow of control to the actor. The control flow inside the actor could be independent of its data communication. An actor may be directly associated with a thread of control, in which case the actor is a process. Nevertheless, it is more common in real-time systems to have a scheduler or an event dispatcher that activates actors by triggers. The hiding of states and the decoupling of data flow and control flow are two major distinctions between Ptolemy actors and general software tasks or processes.

Actor-oriented programming can be conveniently visualized using block diagrams, which are familiar to most control engineers. For example, the embedded system in Figure 1 can be represented by a very similar actor model, as shown in Figure 7. There are two actors: a Supervisor and a Controller. The supervisor has an input port, named data, and an output port, named param. The controller actor has two input ports, input and paramIn, and one output port. These ports decouple the threads of control among the supervisor, the controller, and the scheduler that manages their execution. For example, consider the link from the param port of the supervisor to the paramIn port of the controller. Suppose this link is implemented as a buffer, such that the output from the supervisor is first buffered at the paramIn port, and then the controller decides when to read it (e.g., the next time the controller is activated). Although mutual exclusion between reads and writes on the port is still needed, the controller can always proceed using its internal state without waiting for the supervisor to finish writ-

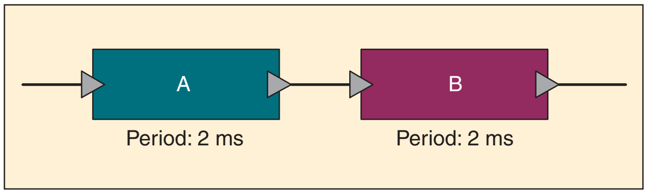


Figure 6. *Two tasks in the Giotto model. Task B always sees data produced by A in the previous cycle.*

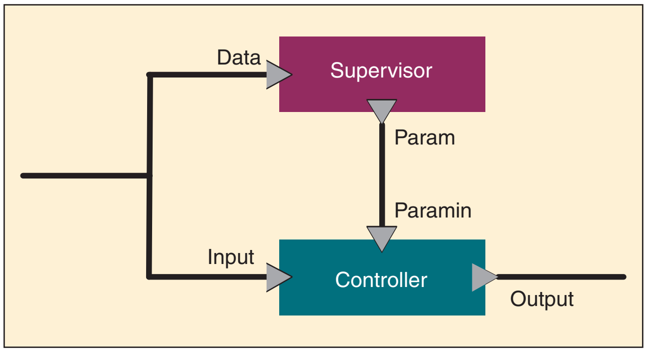


Figure 7. *An actor model for the embedded controller in Figure 1.*

ing and can update parameters at the beginning of the next invocation. The supervisor task can now be arbitrarily preempted by the controller without worrying about priority inversion or about the consistency of parameter data.

The actor architecture can also make resource utilization explicit. A useful design pattern is to use specific actors for delegated resources. For example, suppose that two actors, A and B, both need to write to the same file, as shown in Figure 8. Obviously, an implicit locking mechanism is needed if both of them implement the file writing inside their code. Alternatively, a third actor, fileWriter, can be introduced to manage the file explicitly. By using this actor, the order of writing in actors *A* and *B* becomes visible and is formally managed in the same way as any other events in the system, and the actors no longer need to be partially preemptable.

TM Programming Concepts

The building blocks in the TM model are actors with further annotations. Actors in a TM model not only declare their computing functionality, but also specify their execution requirements in terms of trigger conditions, execution time, and deadlines. As illustrated in Figure 9, an actor is activated when its trigger condition is satisfied. If there are enough resources at run time, then the actor will be granted at least the declared execution time before its deadline is reached. The results of the execution are made available to other actors and the physical world only at the deadline time. This is sometimes called faster-than-real-time computation [17] in the sense that the results are computed well before required by the real-time constraints. Since the output may immediately trigger other tasks, whose deadlines are also known, we effectively control both the starting and stopping times of the actors and provide deterministic timing behavior. In the cases where an actor cannot finish by its deadline, our model includes an overrun handler, much like the timing failure handlers in Chimera [18], to preserve the timing determinism of all other actors and allow the actor that violates the deadline to come to a quiescent state.

Trigger Conditions

A trigger condition can be built using real-time physical events, communication packets, and/or messages from other actors. The trigger conditions are required to be responsible [19], meaning that once triggered, the actor does not need any additional data to complete its (finite) computation. The communication among the actors has an event semantics in which, unlike state semantics, every piece of data will be produced and consumed exactly once. Event semantics can be implemented by FIFO queues. Conceptually, the sender of a communication is never blocked on writing. The responsible trigger condition guarantees that if an actor is activated, there are enough data to complete a reaction so actors will not be blocked on reading data.

Deadlines

The deadline of a reaction is expressed as a real-time value, indicating that the computational results are produced if and only if the deadline is reached. Explicitly expressing deadlines has two benefits:

- By knowing the deadlines of all actors and their triggering dependencies, designers know exactly what time delay their programs will introduce at run time, so that they can be more confident in choosing algorithm parameters.
- Explicit deadlines are also useful in resource-aware algorithms, such as anytime algorithms [20], which can provide results with different fidelities, depending on the computational time available.

If a deadline is not specified, the results will be produced as soon as the computation has finished. This is useful to handle soft real-time actors or intermediate computational steps to provide prompt triggering for downstream actors.

Execution Time

An actor also declares its execution time, which is the amount of time it needs to finish its reaction, provided that it is not preempted by any other actors. Unlike the deadline, which is given in terms of real time no matter how many times the actor is preempted, the execution time is measured with respect to the actual time spent on the actor's execution. This value is mainly used at compile time to determine schedulability and resource utilization. The execution time can be the WCET so that the actor can always finish its reaction. Alternatively, the declared time can be a relaxed typical execution time so that resources may be better used, and the actor can rely on overrun handlers when deadlines are missed.

Overrun Handling

Overrun handlers are nonpreemptable pieces of code that are triggered by the run-time system when the corresponding actors are about to miss deadlines. Since the TM model is event driven and the execution time of reactions varies, it is generally impossible to guarantee that all deadlines of all reactions can be met. By having overrun handlers, the TM model can preserve time determinism as much as possible when resources are not sufficient. How to handle overruns is application dependent. For example, if the actor implements an anytime algorithm, then the overrun handler may terminate the current iteration and bring internal variables to a quiescent state. If the actor implements a transaction-based algorithm, then the overrun handler may commit the transaction if it has finished or may roll back the transaction if it has not finished. If the application is safety critical, then the overrun handler may trigger a mode change into a more conservative mode of operation. In all cases, the actor should be ready for the next activation after running the overrun handler.

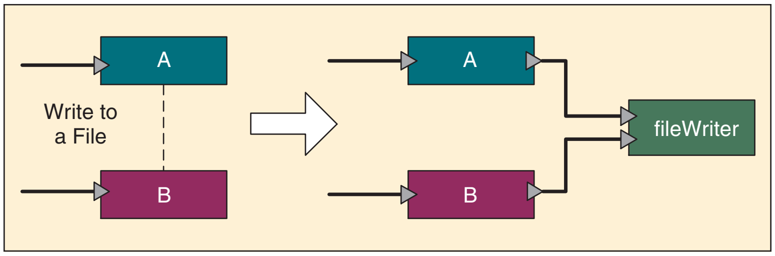


Figure 8. *Shared resources in the TM model can be managed by delegated actors.*

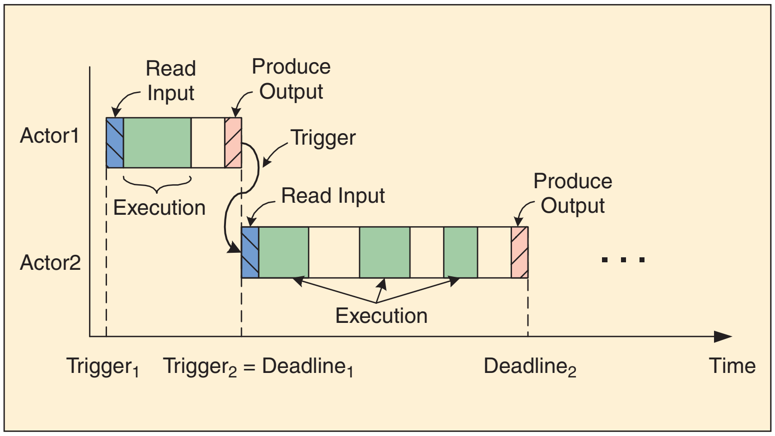


Figure 9. *An illustration of the execution of an actor in the TM model.*

Software Synthesis

Building real-time embedded software using actors and the timed-multitasking model allows certain levels of analysis of timing properties and generation of run-time software and scheduling. This formal step preserves the TM execution semantics and reduces the burden of writing error-prone code for event queues and task synchronizations. We consider a single-processor platform at this time and discuss two aspects in the synthesis: scheduling analysis and code generation.

Scheduling Analysis

General event-triggered, real-time systems with multiple shared resources are not amenable to compile-time schedulability analysis [10]. However, explicit actor partition and time determinism make real-time scheduling theories applicable to many TM models. When trigger conditions are predictable, such as periodic triggers, the execution time and the deadlines can be fed to scheduling algorithms for schedulability analysis and priority assignment. Notice that the semantics of TM does not require that a designer directly specify the priorities of actors. Typically, there are multiple priority assignment policies that can fulfill the timing requirements, and for any feasible scheduling policy, the execution result for a given set of inputs is exactly the same in terms of both time and value determinism. In this sense, the TM model is robust to scheduling policies.

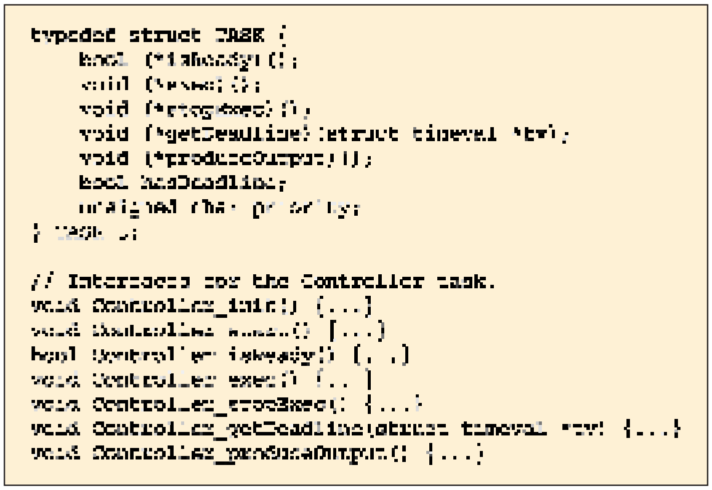
Code Generation

Fully automated software synthesis is a very broad issue. In this article, we focus on generating the interfaces and interactions among actors into imperative languages like C and leave the definition of the actor functionality as an open issue. In fact, after providing the interface and scheduling code, the actor code becomes single threaded and self-contained, which is much easier to manage. The generated code is linked with a TM run-time system, which will be described in the next section.

The first step of software synthesis is to distinguish two types of actors—ones that respond to external events and ones that are triggered entirely by events produced by peer actors. By partitioning software components into actors, it is easy to make sure that these two types do not intersect. At run time, we call the first type interrupt service routines (ISRs) and the second type tasks. For code generation, they have different interfaces.

An ISR typically converts inputs from the outside world into events that trigger other actors. In a TM model, an ISR usually appears as a source actor or simply a port that transfers events into the model. These components do not have triggering rules. They are typically managed by low-level device drivers and are triggered immediately when interrupts occur. An ISR does not have a deadline, and an output is made immediately available as a trigger event to the downstream actors. An ISR is synthesized as an independent thread, with an `init()` method that initializes hardware resources and a `start()` method that registers itself to the run-time system. A `produceOutput()` method is also generated for the ISR to produce trigger events for other actors.

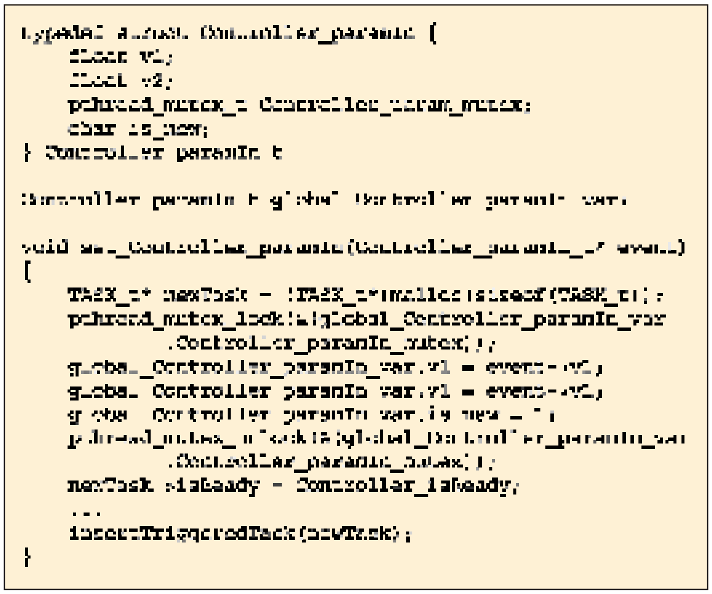
Tasks, on the other hand, have a much richer set of interfaces than ISRs. In additional to `init()` and `start()` methods that initialize resources and states, there is a set of methods, shown in Figure 10, that defines the split-phase reaction of a task, in this case, a Controller. Among the methods, `isReady()` returns a Boolean that indicates if the task is ready to execute when the method is called; `getDeadline()` assigns the deadline value if a deadline is specified; `exec()` is the main body of the reaction; `stopExec()` is the overrun handler; and `produceOutput()` produces events that may trigger other tasks. Obviously, this interface very closely matches the TM programming model. In the interface generation step, for each TM actor that is not an ISR, a C file will be generated that contains the template of all methods listed. A `TASK` data structure is also generated, which defines a scheduling entry representing a task. In addition to the function pointers that point to the task methods, there are two variables: `hasDeadline` is a variable indicating whether the reaction should meet a deadline; `priority` is an integer indicating the ordering among triggered tasks, assuming a static priority assignment.



```
typedef struct TASK {
    bool (*isReady)();
    void (*exec)();
    void (*stopExec)();
    void (*getDeadline)(struct timeval *tv);
    void (*produceOutput)();
    bool hasDeadline;
    unsigned char priority;
} TASK_t;

// Interfaces for the Controller task.
void Controller_init() {...}
void Controller_start() {...}
bool Controller_isReady() {...}
void Controller_exec() {...}
void Controller_stopExec() {...}
void Controller_getDeadline(struct timeval *tv) {...}
void Controller_produceOutput() {...}
```

Figure 10. *Definition of a TASK in generated C code.*



```
typedef struct Controller_paramIn {
    float v1;
    float v2;
    pthread_mutex_t Controller_param_mutex;
    char is_new;
} Controller_paramIn_t

Controller_paramIn_t global_Controller_paramIn_var;

void set_Controller_paramIn(Controller_paramIn_t* event)
{
    TASK_t* newTask = (TASK_t*)malloc(sizeof(TASK_t));
    pthread_mutex_lock(&(global_Controller_paramIn_var
            .Controller_paramIn_mutex));
    global_Controller_paramIn_var.v1 = event->v1;
    global_Controller_paramIn_var.v1 = event->v2;
    global_Controller_paramIn_var.is_new = 1;
    pthread_mutex_unlock(&(global_Controller_paramIn_var
            .Controller_paramIn_mutex));
    newTask->isReady = Controller_isReady;
    ...
    insertTriggeredTask(newTask);
}
```

Figure 11. *Generated code that defines an event type, a port, and a triggering mechanism, which activates the controller task when new events are sent to the port.*

The software synthesis process also generates the interaction relations among actors by sorting through ports and connections. In a TM model, ports are contained by actors. Thus, it is obvious to the code generator which task to trigger when a new event is received by a port. There are many ways to achieve event-triggered execution models. Here we use an event dispatcher at the scheduler level; therefore, ports become proxies for the event dispatcher. Events on the same connection are represented by a global data structure, which contains the communicating data, a mutual-exclusion lock to guard the access to the variable if necessary, and a flag indicating whether the event has been consumed. Figure 11 is an example of the code generated for the port paramIn of the actor Controller in Figure 7, assuming the data type of the port is a pair of floats and assuming an underlying operating system that supports POSIX threads. A data structure for the events is generated, together with a `set_Controller_paramIn()` method that represents putting an event into the paramIn port. The last line in the generated `Supervisor_produceOutput()` template will call the `set_Controller_paramIn()` method with a new event. In this method, the data in the event is transferred to the global variable representing the port, and a task trigger is created and queued with a dispatcher. A run-time scheduler will use the dispatcher and trigger the task at a proper time.

TM Run-Time Systems

The execution model of TM programs is a stylized use of priority-based multitasking execution, as seen in most RTOSs. The run-time scheduler, implemented as a highest priority task, uses a dispatcher to manage trigger events and to execute reactions. The scheduler is activated by an `insertTriggeredTask(TASK_t* task)` call when a port receives a new event. The scheduler then invokes the `isReady()` method of the corresponding task. If the task is ready, the scheduler will send the task's `exec()` to the underlying RTOS to schedule its execution. At the same time, it sets up a timer to monitor the task's deadline, if necessary.

The run-time system for TM programs strictly obeys the deadlines for each actor if they are specified. It keeps track of the deadlines for all actors as timer interrupts, and when the deadline is reached, it asks the actor to stop execution and produce its outputs. If the actor is still executing at the deadline time, the overrun handler will be invoked.

An overrun reaction can be terminated either gracefully or abruptly. A graceful termination may set a flag to ask the reaction to stop and rely on the reaction code to check this flag at reasonable frequencies. The actor can then invoke the overrun handler itself and return from the reaction. An abrupt termination requires the scheduler to terminate the reaction regardless of what is executing and invoke the overrun handler to bring the actor to a safe state. Abrupt terminations may be more efficient in time, but both the scheduler and the overrun handlers may need assembly-level access to tasks' internal state, which makes the run-time systems less portable.

Example

In this section, we describe the design and hardware-in-the-loop (HITL) simulation of an unmanned aerial vehicle (UAV) controller using the TM model. The helicopter UAV belongs to the Berkeley Aerial Robot (BEAR) team, and the control algorithms have been proposed in [21]. The first generation of the flight control software is implemented as a set of hand-tuned tasks on top of an RTOS. Due to the significant loss of modularity, the performance of the controller is very hard to predict. In [22], Horowitz et al. implemented a time-triggered controller in Giotto that greatly improves the modularity and time determinism of the embedded software. However, the asynchrony between sensors and the controller and the timing requirements of the controller do not exactly match the Giotto assumptions. Here we design the same control system in TM, which is much more flexible yet still preserves timing determinism.

Flight Control System

The primary components in the flight control system are actuators, sensors, and a control computer. The actuators consist of servomotors controlling the collective pitch, cyclic pitch, throttle, and tail rotor. The primary sensors are:

- *Inertial navigation system* (INS), which consists of accelerometers and rotational rate sensors and provides estimations of the helicopter's position, velocity, orientation, and rate of rotation. Although the information is provided at a rate of 100 Hz, the error in the estimates could grow unbounded over time.

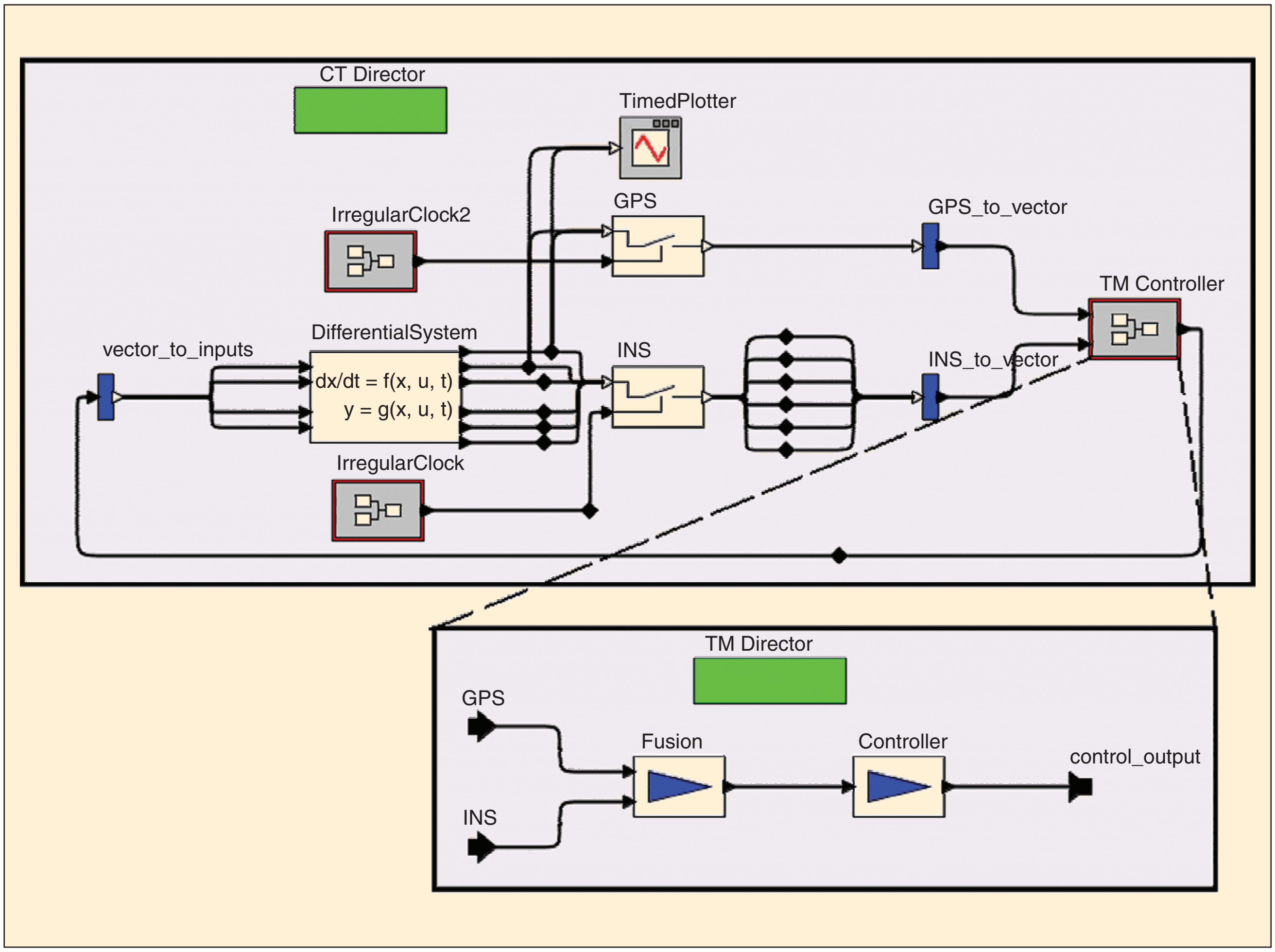


Figure 12. *Ptolemy II model for the flight control system.*

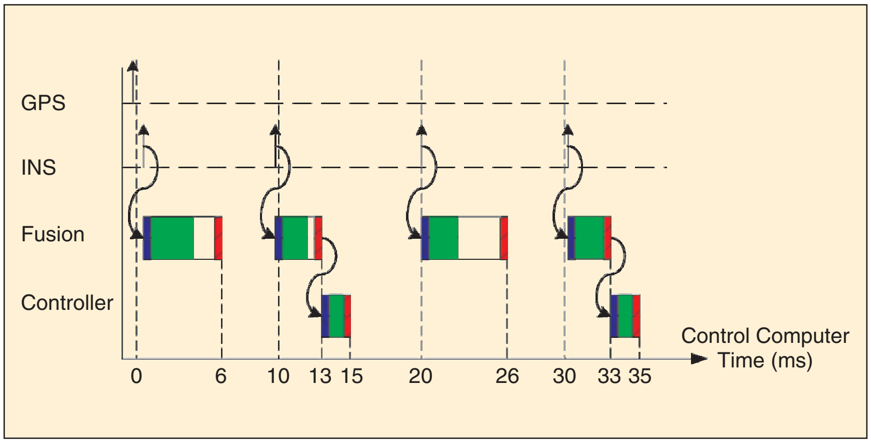


Figure 13. *Timing diagram for the flight control system.*

- *Global positioning system* (GPS), which solves the INS drifting problem by providing a more accurate position measurement, but less frequently, at roughly 5 Hz.

Both sensors push their measurement into the control computer as interrupts. A Kalman filter, implemented in embedded software, fuses the INS-GPS readings to provide frequent and accurate estimates of the state of the helicopter.

The controller implements a modal state feedback algorithm, which partitions the helicopter trajectory into operation modes and uses state feedback to stabilize and drive the helicopter. The controller runs at 50 Hz, which is determined by the helicopter dynamics and the limitations of the actuators. In essence, the controller runs on every second output of the sensor fusion component.

Giotto has two difficulties with the flight control system. First, the clocks of the GPS and INS are not synchronized with the control computer. If sensor fusion is implemented as a Giotto task of frequency 100 Hz, then when the task is triggered according to the control computer clock, the sensor inputs may not be fresh. Second, if the control task is modeled using Giotto as a 50-Hz task, then the controller output will only be available to the actuator by the end of a 20-ms period, even though the control algorithm can be finished in about 2 ms. The latter problem is solved in [22] by designing the controller as a 200-Hz task where in three out of every four cycles, the controller does no computation. Even so, the control system introduces a total delay of 15 ms.

Modeling in TM

TM is a suitable model for the UAV flight control system, since it is event driven, so that no explicit clock synchronization is required between the sensors and the control computer. In addition, the explicit notion of deadline allows the timing properties of the fusion task and the controller to be expressed flexibly. There is no restriction such as that the output has to be produced just before the start of the next period.

The flight control system is modeled in the TM domain of Ptolemy II, and the helicopter dynamics are modeled in the continuous-time (CT) domain. Figure 12 shows the model. The helicopter dynamics are implemented as a set of differential equations using the DifferentialSystem actor. Two triggered samplers and irregular clocks are used to model the unsynchronized outputs from INS and GPS. There are two tasks in the TM Controller—Fusion and Controller. The Fusion actor is triggered by the INS inputs. For the GPS inputs, it consumes values but does not execute the Kalman filter. Thus, the actor is executed at a 100-Hz rate, driven by the INS clock. The Controller is triggered by every other output of the Fusion actor. The timing of the actors is shown in Figure 13. Let k be an integer that increases by one for every 10 ms. Then the deadline of the Fusion actor is $(10k + 6)$ ms in the cycle when the controller is not activated. Otherwise, the deadline is $(10k + 3)$ ms. The deadline of the Controller is always $(20k + 5)$ ms. Having these deterministic deadlines, the controller introduces a fixed delay of 5 ms in every control cycle.

Hardware-in-the-Loop Simulation

The controller is validated further in an HITL simulator implemented in the BEAR project [22]. Interfaces and schedules of the embedded software components are generated from the Ptolemy II model into C code running on Linux. The two input ports of the TM Controller are generated as interrupt service routines. The Fusion and Controller actors are generated into TM run-time tasks. The output port control_output is synthesized into a trivial task. Its `isReady()` method always returns 1 and it does not have a deadline. The execution of this task simply drives the actuators.

The HITL simulator executes the helicopter dynamics and provides sensor and actuator interfaces as UDP datagram sockets. The INS and GPS interrupt service routines are implemented as UDP servers that are executed in separate threads. Received UDP packets are converted into events that trigger the fusion task. Deadline monitors are implemented as separated threads that wake up at the deadline times and call `stopExec()` on corresponding tasks. A graceful task termination policy is implemented where the fusion and the controller execution code checks flags that can be set by a corresponding `stopExec()`. If a deadline is missed, the results from the last cycle are produced. The fusion task uses the spare time in the long cycle to catch up on missed computation. Since the controller, implementing state feedbacks, is stateless, no extra computation is necessary for compensating for missed deadlines.

Conclusion

This article reviewed the challenges of developing embedded software for real-time control systems and argued that timing properties should be introduced at the programming level to bridge the gap between algorithm development and real-time priority tuning. We further described the timed multitasking model that uses an event-triggering mechanism and deadlines to provide deterministic timing behavior for embedded software. The actor-oriented programming model and a highly structured communication style allowed interface and scheduling code to be generated from the high-level specification. A helicopter control system was designed as an example.

Acknowledgments

This work is part of the Ptolemy project, which is supported by DARPA, the State of California MICRO program, and the following companies: Agilent, Cadence Design Systems, Hitachi, and Philips. The authors would also like to thank Judy Liebman for providing the hardware-in-the-loop helicopter simulator.

References

[1] D.B. Stewart, R.A. Volpe, and P.K. Khosla, “Design of dynamically reconfigurable real-time software using port-based objects,” *IEEE Trans. Software Eng.*, vol. 23, pp. 759-776, Dec. 1997.

[2] T.A. Henzinger, B. Horowitz, and C.M. Kirsch, “Embedded control systems development with Giotto,” in *Proc. Languages, Compilers, and Tools for Embedded Systems (LCTES’01)*, Salt Lake City, UT, June 2001, pp. 64-72.

[3] E.A. Lee, “What’s ahead for embedded software,” *IEEE Computer*, vol. 33, pp. 18-26, Sept. 2000.

[4] C. Liu and J. Layland, “Scheduling algorithms for multiprogramming in a hard real-time environment,” *J. ACM*, vol. 10, no. 1, pp. 46-61, 1973.

[5] J.P. Lehoczky, L. Sha, and Y. Ding, “The rate monotonic scheduling algorithm: Exact characterization and average case behavior,” in *Proc. IEEE Real-Time Systems Symp.*, Dec. 1989, pp. 166-171.

[6] N.C. Audsley, A. Burns, M. Richardson, and A. Wellings, “Hard real-time scheduling: The deadline monotonic approach,” in *Proc. IEEE Workshop on Real-Time Operating Systems and Software*, May 1991, pp. 133-137.

[7] K. Tindell, A. Burns, and A. Wellings, “An extendible approach for analyzing fixed priority hard real-time tasks,” *Real-Time Syst. J.*, vol. 6, no. 3, pp. 133-151, 1994.S

[8] M. Spuri and G. Buttazzo, “Scheduling aperiodic tasks in dynamic priority systems,” *Real-Time Syst. J.*, vol. 10, no. 2, pp. 179-210, 1998.

[9] Y. Wang and M. Saksena, “Scheduling fixed-priority tasks with preemption threshold,” in *Proc. Int. Conf. Real-Time Computing Systems and Applications*, Hong Kong, Dec. 1999.

[10] J. Blazewicz, W. Cellary, R. Slowinski, and J. Weglarz, *Scheduling Under Resource Constraints—Deterministic Models* (Ann. Oper. Res., vol. 7). Basel, Switzerland: Baltzer, 1986.

[11] R. Rajkumar, *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Norwell, MA: Kluwer, 1991.

[12] F. Kolnick, *The QNX 4 Real-time Operating System*. Ottawa, Ont., Canada: Basis Computer Systems, 2000.

[13] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa, “Resource kernels: A resource-centric approach to real-time systems,” in *Proc. SPIE/ACM Conf. Multimedia Computing and Networking (MMCN’98),* San Jose, CA, Jan. 1998.

[14] H. Kopetz, “The time-triggered architecture,” in *Proc. 1st Int. Symp. Object-Oriented Real-Time Distributed Computing (ISORC’98)*, Kyoto, Japan, 1998, pp. 22-29.

[15] J. Davis II, C. Hylands, B. Kienhuis, E.A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Tsay, B. Vogel, and Y. Xiong, “Ptolemy II: Heterogeneous concurrent modeling and design in Java,” Tech. Memo., UCB/ERL M01/12, EECS, University of California, Berkeley, CA 94720, Mar. 2001.

[16] G.A. Agha, *ACTORS: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA: MIT Press, 1986.

[17] D. Tennenhouse, “Proactive computing,” *Commun. ACM*, vol. 43, no. 5, pp. 43-50, May 2000.

[18] D.B. Stewart and P.K. Khosla, “Mechanisms for detecting and handling timing errors,” *Commun. ACM*, vol. 40, no. 1, pp. 87-94, Jan. 1997.

[19] J. Liu, “Responsible frameworks for heterogeneous modeling and design of embedded systems,” Ph.D. dissertation, EECS, Univ. of California, Berkeley, Fall 2001.

[20] T. Dean and M. Boddy, “An analysis of time-dependent planning,” in *Proc. 7th Nat. Conf. Artificial Intelligence*, St. Paul, MN, August 1988, pp. 49-54.

[21] D.H. Shim, T.J. Koo, F. Hoffmann, and S.S. Sastry, “A comprehensive study of control design for an autonomous helicopter,” in *Proc. 37th Conf. Decision and Control*, Tampa, FL, 1998, pp. 3653-3658.

[22] B. Horowitz, J. Liebman, C. Ma, T.J. Koo, T.A. Henzinger, A. Sangiovanni-Vincentelli, and S. Sastry, “Embedded software design and system integration for rotorcraft UAV using platforms,” in *Proc. 2002 IFAC World Cong.*, Barcelona, Spain, July 2002 [CD-ROM].

Jie Liu is a member of the research staff at Palo Alto Research Center (formerly Xerox PARC). He received his bachelor’s and master’s degrees in automatic control from Tsinghua University, Beijing, China, in 1993 and 1996, respectively, and his Ph.D. degree in electrical engineering and computer sciences from the University of California, Berkeley, in 2001. His research interests are in modeling, simulation, and design of embedded systems; architectures, languages, and synthesis of embedded software; and embedded system applications. He is a member of the IEEE and ACM. He received the 2001 Leon O. Chua Award from the University of California, Berkeley, for outstanding research on nonlinear science.

Edward A. Lee is a professor in the Electrical Engineering and Computer Science Department at the University of California, Berkeley, where he is also the director of the Ptolemy project. His research interests center on design, modeling, and simulation of embedded, real-time computational systems. He is co-author of five books and numerous papers. His B.S. degree is from Yale University (1979), his S.M. from MIT (1981), and his Ph.D. from the University of California, Berkeley (1986). From 1979 to 1982, he was a member of the technical staff at Bell Telephone Laboratories in Holmdel, New Jersey. He is a cofounder of BDTI, Inc., where he is currently a senior technical advisor, and has consulted for a number of other companies. He is a Fellow of the IEEE, was an NSF Presidential Young Investigator, and won the 1997 Frederick Emmons Terman Award for Engineering Education.