Modeling Event-Based Systems in Ptolemy II

EE249: Design of Embedded Systems: Models, Validation, and Synthesis

Fall 2001

Elaine Cheong and Yang Zhao

Abstract

Networked sensors are becoming an increasingly important field of research as power and size requirements grow smaller and it becomes easier to embed thousands or millions of these devices in our environment. However, there is currently very little programming support for these and other event-based systems. In this paper, we focus on TinyOS, an event-based operating system for Smart Dust networked sensors. We show how to model and simulate TinyOS at the scheduler level in Ptolemy II.

1 Introduction

Advances in digital circuitry, wireless communications, and MEMS (microelectromechanical systems) have led to reductions in size, power consumption, and cost of electronics. This has enabled remarkably compact, autonomous nodes, each containing one or more sensors, computation and communication capabilities, and a power supply. The Smart Dust project [13] at UC Berkeley, led by Profs. Kris Pister and Joseph Kahn, aims to incorporate the requisite sensing, communication, and computing hardware, along with a power supply, in a volume no more than a few cubic millimeters, while still achieving impressive performance in terms of sensor functionality and communications capability. These millimeter-scale nodes are called "Smart Dust" [11][8].

While these researchers investigate how to create the hardware for sensor node systems, other researchers explore software solutions for managing large-scale networks of wireless sensors. The research group of Prof. David Culler at UC Berkeley has developed TinyOS, a tiny event-based operating system for networked sensors [14].

Many other researchers are interested in using TinyOS to build larger applications for networked sensors [15]. However, concurrent interactions between TinyOS components can make TinyOS applications very difficult to understand and code correctly. Moreover, TinyOS applications are extremely difficult to debug once they are deployed onto the target platform.

The Ptolemy project [12], led by Prof. Edward Lee at UC Berkeley, studies modeling, simulation, and design of concurrent, real-time, embedded systems. The focus is on assembly of concurrent components. The key underlying principle in the project is the use of well-defined models of computation that govern the interaction between components. A major problem area being addressed is the use of heterogeneous mixtures of models of computation. Ptolemy II is a set of Java packages supporting heterogeneous modeling, simulation, and design of concurrent systems.

The goal of our project is to use the Ptolemy II modeling environment to model the interaction between TinyOS components at the scheduler level. This project forms the basis for modeling larger TinyOS applications within Ptolemy II and eventually generating application code directly from the model. In this manner, a TinyOS application developer can fully understand and debug the application in a desktop simulation environment before testing on the actual embedded platform, which can be extremely time consuming and error-prone.

2 TinyOS

Networked sensors must react to the real-world environment, which is inherently concurrent. We first introduce the two main ways of structuring concurrent systems, as described by Welsh et al. [16]: *thread-driven* and *event-driven*.

In the *thread-driven* (*thread-based*) approach, threads and processes are primarily designed to support multiprogramming, and existing operating systems strive to virtualize hardware resources in a way that is transparent to applications. The system uses a main thread which hands off tasks to individual task-handling threads, which step through all the stages of processing that task. Threads are the standard form of expressing concurrency in most operating systems, and tools for programming in the sequential style of threads are relatively mature. However, there are several problems with thread programming. Synchronization primitives (such as locks, mutexes, or condition variables) are a common source of bugs, and lock contention can cause serious performance degradation as the number of threads competing for a lock increases. Additionally, the overhead of creating a handler thread for each task remains, although creating a pool of threads in advance can mitigate this cost. Finally, context switching between threads incurs a high overhead.

In the *event-driven* (*event-based*) approach, the system uses a single thread and non-blocking interfaces to I/O subsystems or timer utilities to "juggle" between concurrent tasks. The processing of each task is implemented as a finite state machine, where transitions between states in the FSM are triggered by events. Event-driven programming has its own problems. Separate threads no longer handle the sequential flow of each task; instead, one thread processes all tasks in disjoint stages. This can make debugging difficult, as stack traces no longer represent the control flow for the processing of a particular task. Also, task state must be bundled into the task itself, rather than stored in local variables or on the stack in a threaded system. Event-packages are not standardized, and there are few debugging tools for event-driven programming. However, event-driven programming avoids many of the problems associated with synchronization, such as race conditions and deadlocks. Additionally, event-based systems can be faster than thread-based systems, since they avoid the overhead of thread creation and context switching.

TinyOS [7] is a tiny event-based operating system for networked sensors designed for efficient modularity and concurrency-intensive operation. Since networked sensor hardware can vary widely depending on the particular

application, efficient modularity allows the operating system to be customized to fit application-specific needs. TinyOS runs on networked sensors must be able to respond to concurrent events from the real-world environment and process the data quickly.

TinyOS is written in the C programming language and uses C preprocessing macros to simulate an object-oriented programming language, but with less overhead.

In the following sections, we describe the TinyOS component model, scheduler, and application structure in detail.

2.1 TinyOS Component Model

A TinyOS application consists of a graph of components. Components are arranged hierarchically. Low-level components map physical hardware into the component model. High-level components perform control, routing, and all data transformations. A component has four parts: a *frame*, a set of *command handlers*, a set of *event handlers*, and a bundle of *tasks*.

A frame is a statically allocated range of memory associated with each component. A component stores its state in this fixed-sized frame.

A command is a non-blocking request made to a lower-level component. A command handler is a routine associated with the lower-level component that runs in response to the command. A command handler provides feedback to its caller by returning success/failure status. In essence, a command is a C function call.

In TinyOS, an event is the occurrence or happening of significance to the program, such as the completion of an asynchronous input/output operation. All events in TinyOS originate from hardware interrupts. An event handler is a routine invoked by the operating system to deal with hardware events, either directly or indirectly.

A task represents a long-running computation. Tasks are atomic with respect to other tasks and run to completion, though they can be preempted by events. Tasks provide a way to incorporate arbitrary computation into the event-driven model.

The calling relationship between commands, events, and tasks is summarized below:

- Commands may call other commands or post tasks, but may not signal events.
- Events may call commands, post tasks, or signal other events.
- Tasks may call commands, post other tasks, or signal events.

Next, we discuss the scheduling of commands, events, and tasks in TinyOS.

2.2 TinyOS Scheduling

TinyOS uses a two-level scheduler, in which a small amount of processing associated with hardware events are performed immediately, while long running tasks may be interrupted. These tasks are the microthreads described in [7], where the authors identify TinyOS as a “microthreaded” operating system. Microthreads are also known as non-blocking threads or non-preemptive threads.

Scheduling of computation in TinyOS is split into two levels. The first level consists of events and commands. These short computations are executed immediately. The second level consists of tasks, which are long-running computations. When a task is “posted”, it is placed in a first-in, first-out (FIFO) queue. Tasks are executed when there are no events or commands to be run. There is a single stack in the system, which is assigned to the currently executing task. A running task may be preempted by events, in which case the interrupt hardware takes care of saving the state of a task. A task, however, may not preempt other tasks.

Next, we discuss how to build and run TinyOS applications.

2.3 TinyOS Applications

A TinyOS application developer first determines how to divide the application into components. For each component, the developer must create two files: a `.comp` file and a `.c` file. Additionally, the developer must create a `.desc` file for the application.

The component file (`.comp`) specifies the interface to a component. This file contains six main parts: (1) the name of the component, (2) the command handlers, or commands that other components can call on this component, (3) the signals, or events that this component generates, (4) the event handlers, (5) the commands that this component calls, (6) and internal functions that are only called by this component.

The source file (`.c`) contains the actual functionality of the component. The source file contains a fixed-size storage frame, along with the code for any command handlers, event handlers, or tasks.

The description file (`.desc`) specifies how to link the components in the application together. For example, if a component calls a command, the developer needs to specify which component’s command handler will run in response. If a component signals an event, the developer must specify which component will handle this event.

The TinyOS scheduler makes several assumptions about components. Commands must not wait for long or indeterminate latency (i.e., non-blocking) actions to take place. A command is intended to perform a small, fixed amount of work, which occurs within the context of its component's state. The same assumptions about commands also apply to event handlers. Tasks perform the primary work. Tasks must never block or spin wait or they will prevent progress in other components.

After specifying the interfaces for all of the components, writing the source code, and determining how to link the components together, the TinyOS application developer has several compilation options: the developer can choose to compile the application for desktop simulation, or he/she can choose to compile the application to a form suitable for downloading onto the embedded hardware [9].

Currently, the desktop simulator runs in a command-line shell. The user can trace items ranging from clock interrupts to LED output to simulator internals. There is also a simulator visualization graphical user interface (GUI) written in Java. However, this graphical tool currently only supports visualization of radio packet messages (incoming and outgoing).

3 Ptolemy II

Programming a TinyOS application can be quite complex. To change a single connection between two components, an

application developer may have to change up to three files. Because TinyOS is event-based, it is difficult to trace the program flow of a task, since it may constantly be interrupted by events and commands. Existing tools for debugging and simulating TinyOS applications are very limited. A graphical simulation environment like Ptolemy II would be a better solution to this problem.

Ptolemy II [3] is a Java software system for construction and execution of concurrent models.

Modeling is the act of representing a system or subsystem formally. A *constructive model* defines a computational procedure that mimics a set of properties of the system. Constructive models are often used to describe behavior of a system in response to stimulus from outside the system. Constructive models are also called *executable models*. Executable models are sometimes called *simulations*, an appropriate term when the executable model is clearly distinct from the system it models. However, in many electronic systems, a model that starts as a simulation mutates into a software implementation of the system. The distinction between the model and the system itself becomes blurred in this case. This is particularly true for embedded software.

Executable models are constructed under a *model of computation*, which is the set of "laws of physics" that govern the interaction of components in the model. For embedded systems, the most useful models of computation handle concurrency and time. This is because embedded systems consist typically of components that operate simultaneously and have multiple simultaneous sources of stimuli. In addition, they operate in a timed (real world) environment, where the timeliness of their response to stimuli may be as important as the correctness of the response.

The objective in Ptolemy II is to support the construction and interoperability of executable models that are built under a wide variety of models of computation. Ptolemy II takes a component view of design, in that models are constructed as a set of interacting components. A model of computation governs the semantics of the interaction, and thus imposes a discipline on the interaction of components.

3.1 Models of computation

A model of computation consists of the rules that govern the interaction, communication, and control flow of a set of components. In Ptolemy II, a *domain* is an implementation of a particular model of computation. Each component is represented as an *actor*, which is an executable entity or node in the component graph. A *director* is an object that controls the execution of the actors according to some model of computation.

In this section, we present three domains that we use in Section 4 to model TinyOS. These include the DE (discrete event), TM (timed multitasking) and FSM (finite state machine) domains.

3.1.1 DE (Discrete Event)

In the DE (discrete event) domain of Ptolemy II, an event is something that carries a timestamp and possibly a value. Actors communicate via sequences of events placed in time, along a real time line. Actors can either be processes that react to events (implemented as Java threads) or functions that fire when new events are supplied. A DE scheduler ensures that events are processed chronologically according to their time stamps by firing those actors whose available input events have the earliest time stamp of all pending events.

In the DE model of computation, time is global, in the sense that all actors share the same global time. The current time of the model is often called the *model time* or *simulation time* to avoid confusion with current real time.

As in most Ptolemy II domains, actors communicate by sending *tokens* through *ports*. Ports can be input ports, output ports, or both. Tokens are sent by an output port and received by all input ports connected to the output port through *relations*. A relation is an object representing an interconnection between entities. When a token is sent from an output port, it is packaged as an event and stored in a global event queue. By default, the time stamp of an output is the model time, although specialized DE actors can produce events with future time stamps.

Actors may also request that they be fired at some time in the future by calling the *fireAt()* method of the director. This places a pure event (one with a time stamp, but no data) on the event queue. A pure event can be thought of as setting an alarm clock to be awakened in the future. Sources (actors with no inputs) can thus be fired despite having no inputs to trigger a firing. Moreover, actors that introduce delay (outputs have larger time stamps than the inputs) can use this mechanism to schedule a firing in the future to produce an output.

In the global event queue, events are sorted based on their time stamps. An event is removed from the global event queue when the model time reaches its time stamp, and if it has a data token, then that token is put into the destination input port.

At any point in the execution of a model, the events stored in the global event queue have time stamps greater than or equal to the model time. The DE director is responsible for advancing (i.e. incrementing) the model time when all events with time stamps equal to the current model time have been processed (i.e. the global event queue only contains events with time stamps strictly greater than the current time). The current time is advanced to the smallest time stamp of all events in the global event queue.

3.1.2 TM (Timed Multitasking)

The TM (timed multitasking) domain implements a model of computation based on priority-driven multitasking, which is common in real-time operating systems (RTOSs), but with more deterministic behavior. The TM domain is a new domain in Ptolemy II, and was formerly known as the RTOS domain.

In the TM domain, actors (conceptually) execute as concurrent threads in reaction to inputs. The domain provides an event dispatcher, which maintains a prioritized event queue. The execution of an actor is triggered by the event dispatcher by invoking first its *prefire()* method. The actor may begin execution of a concurrent thread at this time. Some time later, the dispatcher will invoke the *fire()* and *postfire()* methods of the actor (unless *prefire()* returns false).

The amount of time that elapses between the invocation of *prefire()* and *fire()* depends on the declared *executionTime* and *priority* of the actor (or more specifically, of the port of the port receiving the triggering event). The domain assumes there is a single resource, the CPU, shared by the execution of all actors. At any particular time, only one of the actors can get the resource and execute. Another eligible actor with a higher priority input event may preempt execution of another actor. If an actor is not preempted, then the amount of time that elapses between *prefire()* and *fire()* equals the declared *executionTime.* If the actor is preempted, then the elapsed time equals the sum

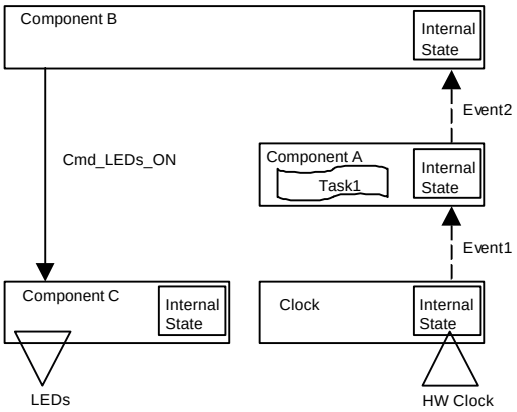

Figure 1: A sample application

of the *executionTime* and the execution times of the actors that preempt it. The model of computation is more deterministic than the usual priority-driven multitasking in RTOSs because the actor produces outputs (in its *fire()* method) only after it has been assured access to the CPU for its declared *executionTime*.

3.1.3 FSM (Finite State Machine)

In the FSM (finite state machine) domain, entities represent state instead of actors, and the connections represent transitions between states. Execution is a strictly ordered sequence of state transitions. Evaluation of the guards on each transition determines when state transitions can be taken.

The FSM domain in Ptolemy II can be hierarchically combined with other domains [5]. The resulting formalism is called "*charts" (pronounced "starcharts") where the star represents a wildcard. Since most other domains represent concurrent computations, *charts model concurrent finite state machines with a variety of concurrency semantics.

4 Modeling TinyOS in Ptolemy II

We have now introduced TinyOS and the problems involved with programming and debugging TinyOS applications. We have also presented Ptolemy II, a software package for heterogenous modeling, simulation, and design of concurrent systems. In order to model and simulate full TinyOS applications, we must first start modeling at the scheduler level.

In this section, we describe how to model the event, command, and task interaction of TinyOS in Ptolemy II by way of example. We first show a sample application. We also explain the basic way we model components in Ptolemy II. We then explain how to use this basic model in the DE and TM domains.

4.1 Sample Application

Figure 1 shows a sample TinyOS application.

In this application, the hardware clock (*HW Clock*) will trigger the *Clock* component, which in turn signals *Event1* to Component A. The *Event1* handler of Component A responds to *Event1* by posting *Task1* and signaling *Event2* to Component B. The *Event2* handler of Component B issues an *LEDs_ON* command down to Component C. The command handler of Component C controls the blinking of the LEDs. This simple example contains most of the features that TinyOS applications can have.

TinyOS uses a component model to achieve efficient modularity. We model a basic TinyOS component with a *TypedCompositeActor* in Ptolemy II. We model a component's event handler or command handler with an *FSMActor* inside of the *TypedCompositeActor*, since the FSM domain can model program logic easily. We can model a task with an actor that depends on the computation of the task. TinyOS assumes there is a single CPU and that the execution context is shared by the components. For example, suppose an event handler *A* signals an event, which is handled by event handler *B*. TinyOS will context switch from *A* to *B*. *B* may take a very short time to complete and will eventually return the execution context to *A*. In Ptolemy, there is currently no domain supporting this function call return feature. To model this, we add a feedback connection from the "callee" actor to the "caller" actor (see Figures 2a and 3a).

We have not yet specified which domain we should use to model TinyOS at the top level, although we have specified that we will use the FSM domain to model the program logic of event and command handlers. We have selected DE and TM as the most suitable existing domains of Ptolemy II. Since TinyOS is event-based, and clock interrupts and message arrival are discrete events, it seems most natural to use a discrete event model of computation. However, we will see that a priority-driven multitasking model of computation like TM provides a more convenient way of modeling task preemption.

We first explain how to model TinyOS in the DE domain. Then, we explain how to model TinyOS in the TM domain.

4.2 Modeling TinyOS in DE

To model TinyOS in the DE (discrete event) domain of Ptolemy II, we have to tackle two problems. First, we must determine how to model the "two-level" scheduler, which schedules

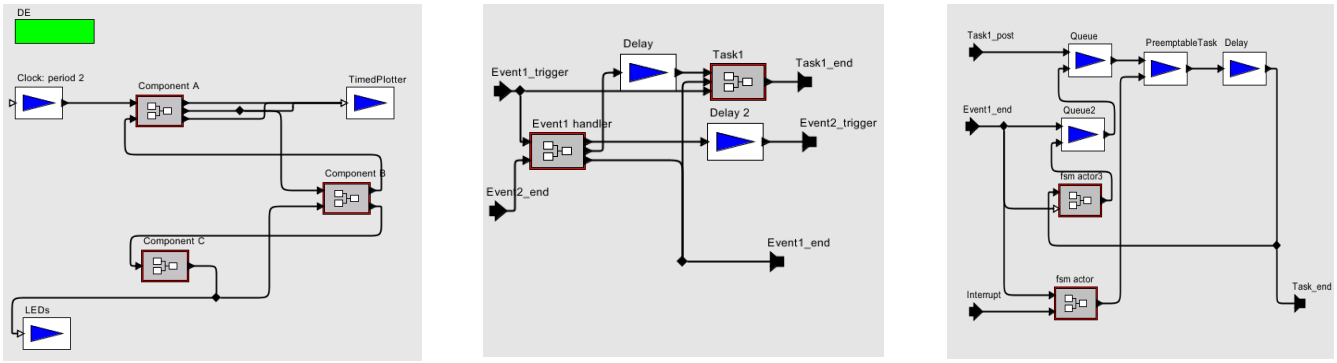

Figure 2: Model of sample application using the DE domain of Ptolemy II. (a) Top level (b) Component A (c) Task1

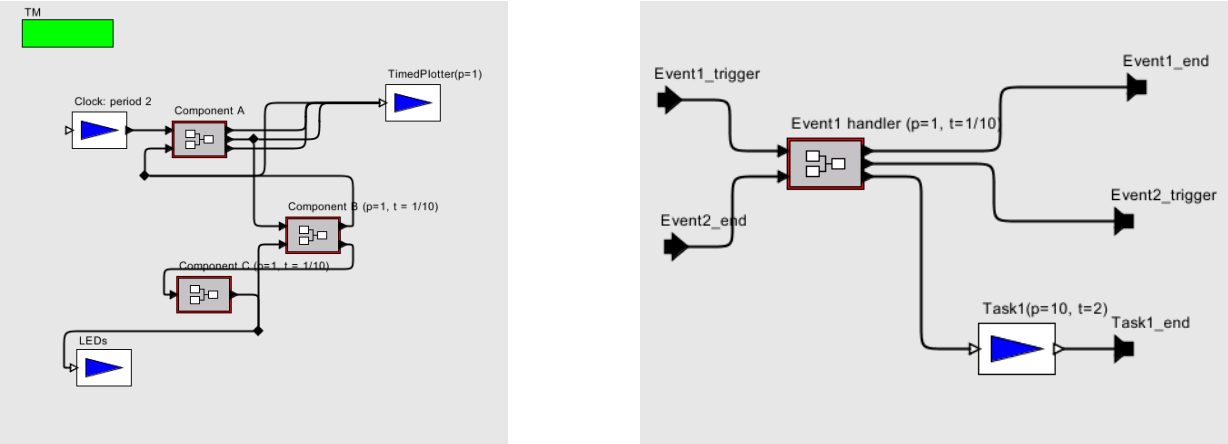

Figure 3: Model of sample application using the TM domain of Ptolemy II. (a) Top Level (b) Component A

events and commands immediately and keep tasks waiting until all events and commands have completed. Second, we must determine how to model the preemption of a task.

The first problem is easier to solve. When an actor posts a task, it will emit a token representing the task. We call this token a *task token*. We do not send this token directly to the corresponding task actor. Instead, we put the task token in a queue and dispatch the token to the task actor when it is the task's turn to run. The queue will only emit the stored task token when there are no events, commands or higher priority tasks (tasks that come earlier in a TinyOS scheduling queue) to execute.

The second problem is difficult to solve with existing Ptolemy actors because of several problems. In the DE domain, an actor fires as soon as data is available on its ports, and logically, actors may fire simultaneously. As stated in the Section 4.1, we would like to use separate actors to model tasks, events, and commands. Event or command actors will not block a task actor from running if its input ports are enabled. Additionally, the DE director will not count the extra time needed by the task if it is interrupted during its execution.

So, to handle the preemption of a task, we build a new actor, which we call *PreemptableTask*. This new actor has two input ports: *input* and *interrupt*. The *input* port receives task tokens, and the *interrupt* port listens to interrupts from events or commands. The *PreemptableTask* actor also maintains a parameter called *executionTime*, which specifies the time it takes to execute the task. When a token is received on the *input* port, the value is stored in the actor and the actor is scheduled to fire at *executionTime* time units later (using the *fireAt()* method). During the execution, if the *interrupt* port receives a token with a value equal to `true`, the task is "preempted". Later, when the value of the token on the *interrupt* port becomes `false`, the task "resumes execution" and the elapsed time is calculated and added to the execution time of the task. The actor is scheduled to fire at this new execution time. The saved input value is emitted as a token when the new execution time has passed.

We use one or more *FSMActors* inside of the DE model in order to model the component logic.

Figure 2a shows the top level of the Ptolemy model of the sample application shown in Figure 1. Figure 2b shows the refinement of Component A, which is a *TypedCompositeActor*. Component A consists two composite actors: *Event1 handler* and *Task1*. *Event1 handler* is an *FSMActor* modeling the logic of the event handler that responds when *Event1* is signaled. *Task1* models the task inside Component A. Figure 2c shows the refinement of *Task1* from Figure 2c. The queues (*Queue* and *Queue2*) and FSM actors (*fsm actor* and *fsm actor3*) on the left side of Figure 2c are used to control the time to dispatch the task token to the *PreemptableTask* actor (*Task1* in Figure 2c). The delay actor has a delay of 0.0, which helps to avoid the zero-delay loop problem characteristic of the discrete event model of computation [5].

As you may have noticed, it is not very convenient to model a task in the DE domain of Ptolemy even for such a simple application. In our example, we need six additional actors to model the refinement of *Task1*. In the following section, we discuss our work with the TM domain, which is more suitable for modeling TinyOS.

4.3 Modeling TinyOS in TM

The TM (timed multitasking) domain of Ptolemy maintains a prioritized event queue, which sorts events according to their priority instead of as in DE, where the director sorts events according to their time stamps. In the TM domain, actors with higher priority may preempt the execution of an actor with lower priority. If an actor is preempted, it will take some extra time, equal to the interruption time, to complete. Each actor in this domain may have a *priority* parameter and an *executionTime* parameter. An actor's input ports can also be specified with the *executionTime* parameter. By giving higher priority to event handler actors and command handler actors, and giving lower priority to task actors, we can easily model the TinyOS features of two level scheduling and preemption of tasks.

Figure 3a shows the top level of the model of the same sample application (Section 4.1) modeled in the DE domain (Section 4.2). This time we model it in the TM domain. Figure 3b shows what is inside of the composite actor of Component A. *Event1 handler* is also an FSMActor modeling the logic of the *Event1* event handler, similar to our DE model. However, in the TM domain, we can easily model the task using a single actor, unlike the DE domain.

By setting each task actor in the TM domain to different priorities, we can have a more sophisticated priority-based scheduler for tasks rather than the simple FIFO scheduler currently used in TinyOS.

We have now shown how to model TinyOS tasks, events handlers, and command handlers in Ptolemy II and how to

connect these models in order to properly model their interaction.

5 Related Work

Freddy Mang and Profs. Luca de Alfaro and Tom Henzinger have been researching how to use interface automata [1] to model the event and command interface of TinyOS using the Interface Automata (IA) domain of Ptolemy II. The interface automata formalism specifies the temporal aspects of software component interfaces. Specifically, an automata-based language captures both input assumptions about the order in which the methods of a component are called, and output guarantees about the order in which the component calls external methods. The complex interaction between events and commands has required some modification of the interface automata semantics, which is currently being researched by Mang [10].

We believe that their work differs from our project in that they are using a more denotational approach, as opposed to our operational approach, in which we try to model the actual execution of TinyOS components. Additionally, Mang et al. do not consider the interaction of tasks in their model.

6 Conclusions and Future Work

TinyOS is a tiny event-based operating system for networked sensors. Programming tools for these and other event-based systems are quite poor. Modeling and simulating event-based systems in a graphical modeling environment on a desktop computer can make programming and debugging event-driven systems much easier. We show how to model TinyOS event handlers, command handlers, and tasks and their interactions in Ptolemy II, an environment for heterogeneous modeling, simulation, and design of concurrent systems.

There remains more exciting work to build upon this project. Our project forms the groundwork for modeling larger TinyOS applications in Ptolemy II. In the future, we hope to generate TinyOS code directly from a Ptolemy II model, which can be compiled and downloaded to the embedded target for running.

While working with Ptolemy II, we have found several areas that need more work in the future. As mentioned in Section 4.1, there are no currently existing domains in Ptolemy II that support function call returns. It may be worthwhile to create a new domain to support the call-return feature or incorporate it into an existing domain. We also plan to improve the *PreemptableTask* actor by building queues into the actor so that external queues will no longer be necessary.

7 Acknowledgements

We would like to thank our mentors Xiaojun Liu and Prof. Edward A. Lee for their invaluable guidance and assistance. We would also like to thank Rob Szewczyk for his help with TinyOS.

8 References

[1] Luca de Alfaro and Thomas A. Henzinger. Interface Automata. *Proceedings of the Ninth Annual ACM Symposium on Foundations of Software Engineering (FSE 2001).*

[2] David E. Culler, Jason Hill, Philip Buonadonna, Robert Szewczyk, and Alec Woo. A Network-Centric Approach to Embedded Software for Tiny Devices. *EMSOFT 2001*, Lecture Notes in Computer Science, Volume 2211, pp. 114-130, 2001.

[3] John Davis II, Christopher Hylands, Bart Kienhuis, Edward A. Lee, Jie Liu, Xiaojun Liu, Lukito Muliadi, Steve Neuendorffer, Jeff Tsay, Brian Vogel, and Yuhong Xiong, "Heterogeneous Concurrent Modeling and Design in Java," *Technical Memorandum UCB/ERL M01/12*, EECS, University of California, Berkeley, March 15, 2001.

[4] Getting up and running with TinyOS. http://webs.cs.berkeley.edu/tos/

[5] A. Girault, B. Lee, E. A. Lee. Hierarchical finite state machines with multiple concurrency models. IEEE *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol.18, (no.6), IEEE, June 1999. p.742-60.

[6] Jason Hill. *A Software Architecture Supporting Networked Sensors*. Masters thesis, December 2000.

[7] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, Kristofer Pister. System architecture directions for network sensors. *ASPLOS 2000*.

[8] J. M. Kahn, R. H. Katz and K. S. J. Pister, "Mobile Networking for Smart Dust". *ACM/IEEE Intl. Conf. on Mobile Computing and Networking (MobiCom 99)*, Seattle, WA, August 17-19, 1999.

[9] Macro-Motes. http://www-bsac.EECS.Berkeley.EDU/~shollar/macro_motes/macromotes.html

[10] Freddy Mang. Personal communication. 20 Nov 2001.

[11] K. S. J. Pister, J. M. Kahn and B. E. Boser, "Smart Dust: Wireless Networks of Millimeter-Scale Sensor Nodes", *Highlight Article in 1999 Electronics Research Laboratory Research Summary*.

[12] The Ptolemy Project. http://ptolemy.eecs.berkeley.edu/

[13] Smart Dust: Autonomous sensing and communication in a cubic millimeter. http://robotics.eecs.berkeley.edu/~pister/SmartDust/

[14] TinyOS: An operating system for Networked Sensors. http://tinyos.millennium.berkeley.edu/

[15] TinyOS Programming Tutorial Workshop. 17 Oct 2001, Berkeley, CA. http://webs.cs.berkeley.edu/bootcamp.html

[16] Matt Welsh, Steven D. Gribble, Eric A. Brewer, and David Culler. A Design Framework for Highly Concurrent Systems. *CS Technical Report No. UCB/CSD-00-110.*