Viptos: A Graphical Development and Simulation Environment for TinyOS-based Wireless Sensor Networks

Elaine Cheong
Edward A. Lee
Yang Zhao

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2006-15
http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-15.html

February 15, 2006

Acknowledgement

This work was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF award \#CCR-0225610), the State of California Micro Program, and the following companies: Agilent, DGIST, General Motors, Hewlett Packard, Infineon, Microsoft, and Toyota.

Viptos: A Graphical Development and Simulation Environment for TinyOS-based Wireless Sensor Networks

Elaine Cheong
Department of Electrical Engineering
and Computer Sciences
University of California, Berkeley
Berkeley, CA 94720 USA
Email: celaine@eecs.berkeley.edu

Edward A. Lee
Department of Electrical Engineering
and Computer Sciences
University of California, Berkeley
Berkeley, CA 94720 USA
Email: eal@eecs.berkeley.edu

Yang Zhao
Department of Electrical Engineering
and Computer Sciences
University of California, Berkeley
Berkeley, CA 94720 USA
Email: ellen_zh@eecs.berkeley.edu

***Abstract*— We describe Viptos (Visual Ptolemy and TinyOS), an integrated graphical development and simulation environment for TinyOS-based wireless sensor networks. TinyOS is a component-based, event-driven runtime environment designed for wireless sensor networks. Viptos allows networked embedded systems developers to construct block and arrow diagrams to create TinyOS programs from any standard library of TinyOS components written in nesC, a C-based programming language. Viptos automatically transforms the diagram into a nesC program that can be compiled and downloaded from within the graphical environment onto any TinyOS-supported target platform. Viptos is built on Ptolemy II, a modeling and simulation environment for embedded systems, and TOSSIM, an interrupt-level discrete event simulator for homogeneous TinyOS networks. In particular, Viptos includes the full capabilities of VisualSense, a Ptolemy II environment that can model communication channels, networks, and non-TinyOS nodes. Viptos extends the capabilities of TOSSIM to allow simulation of heterogeneous networks. Viptos provides a bridge between VisualSense and TOSSIM by providing interrupt-level simulation of actual TinyOS programs, with packet-level simulation of the network, while allowing the developer to use other models of computation available in Ptolemy II for modeling the physical environment and other parts of the system. This framework allows application developers to easily transition between high-level simulation of algorithms to low-level implementation and simulation. This paper presents our experiences with integrating the nesC/TinyOS/TOSSIM and Ptolemy II programming and execution models.**

I. Introduction

Wireless sensor networks provide a way to create flexible, tetherless, automated data collection and monitoring systems. Building sensor networks today requires piecing together a variety of hardware and software components, each with different design methodologies and tools, making it a challenging and error-prone process. Typical networked embedded system software development may require the design and implementation of device drivers, network stack protocols, scheduler services, application-level tasks, and partitioning of tasks across multiple nodes. Little or no integration exists among the tools necessary to create these software components, mostly because the interactions between the programming models are poorly understood. In addition, these tools typically have little infrastructure for building models and interactions that are not part of their original scope or software design paradigms. The goal of this work is to create integrated tools for networked embedded application developers to model and simulate their algorithms and quickly transition to testing their software on real hardware in the field, while allowing them to use the programming model most appropriate for each part of the system.

We choose to focus on TinyOS [1], an open-source runtime environment designed for sensor network nodes known as *motes*, as our underlying programming platform. TinyOS has a large user base – over 500 research groups and companies use TinyOS on the Berkeley/Crossbow motes. It has been ported to over a dozen platforms and numerous sensor boards, and new releases see over 10,000 downloads. TinyOS differs from traditional operating system models in that events drive the behavior of the system. Using this type of execution, battery-operated nodes can preserve energy by entering sleep mode when no interesting events are happening.

A TinyOS program consists of a graph of components that are written in an object-oriented style using nesC [2], an extension to the C programming language. TOSSIM [3], a TinyOS simulator for the PC, can execute nesC programs designed for a mote. TOSSIM contains a discrete event simulation engine which allows modeling of various hardware and other interrupt events. Although a large community uses TinyOS in simulation to develop and test various algorithms and protocols, they face some key limitations when using the nesC/TinyOS/TOSSIM programming tool suite. Users may choose from a few built-in radio connectivity models in TOSSIM, but it is difficult to use other models. TOSSIM can efficiently model large homogeneous networks where the same nesC code is run on every simulated node, but does not allow simulation of networks that contain different programs. Additionally, whereas a TinyOS program consists of a graph of mostly pre-existing nesC components, users must write their programs in a multi-file, text-based format, even though a graphical block diagram programming environment would be much more intuitive.

To address these problems, we turn to VisualSense [4], a Ptolemy II-based graphical modeling and simulation framework for wireless sensor networks that supports actor-oriented definition of sensor nodes, wireless communication channels, physical media such as acoustic channels, and wired subsystems. VisualSense does not include a mechanism for transitioning from a sensor network application developed within the framework to an implementation for real hardware without rewriting the code from scratch for the target platform.

Integrating TinyOS and VisualSense combines the best of both worlds. TinyOS provides a platform that works on real hardware with a library of components that implement low-level routines. VisualSense provides a graphical modeling environment that supports hierarchical, heterogeneous systems. In this paper, we describe our experiences with integrating the programming and execution models and the component libraries of these two systems, necessary for building an integrated tool chain for designing, simulating, and deploying sensor network applications.

Section II provides detailed background information on nesC, TinyOS, TOSSIM, and VisualSense. Section III describes the architecture of the integrated TinyOS and VisualSense toolchain and investigates the semantics of this interface. Section IV presents related

```
configuration SenseToLeds {
} implementation {
  components Main, SenseToInt,
    IntToLeds, TimerC,
    DemoSensorC as Sensor;
  Main.StdControl -> SenseToInt;
  Main.StdControl -> IntToLeds;
  SenseToInt.Timer ->
    TimerC.Timer[unique("Timer")];
  SenseToInt.TimerControl ->
    TimerC;
  SenseToInt.ADC -> Sensor;
  SenseToInt.ADCControl ->
    Sensor;
  SenseToInt.IntOutput ->
    IntToLeds;
}
```

(a)

```
module SenseToInt {
  provides {
    interface StdControl;
  }
  uses {
    interface Timer;
    interface StdControl
        as TimerControl;
    interface ADC;
    interface StdControl
        as ADCControl;
    interface IntOutput;
  }
} implementation {
    ...
}
```

(b)

Fig. 1. Sample nesC source code.

work. Sections V and VI conclude with a discussion of work-in-progress and future work.

II. Background

In this section we provide detailed information on the nesC syntax, TinyOS execution model, TOSSIM architecture, and VisualSense framework. We feel that this background material is essential to understanding the contributions of our work and include material previously discussed in [4].

A. nesC

A TinyOS program consists of a set of nesC components that are “wired” together. Figure 1a shows a TinyOS program called `SenseToLeds` that displays the value of a photosensor in binary on the LEDs of a mote. TinyOS includes a library of nesC components, including the ones listed in `SenseToLeds`, such as `Main`, `SenseToInt` (shown in Figure 1b), `IntToLeds`, `TimerC`, and `DemoSensorC`.

A nesC component exposes a set of *interfaces*. An interface consists of a set of methods. A method is known as either a *command* or an *event*. The component implements its *provides* methods and expects another component to implement its *uses* methods. nesC interfaces can also be *parameterized* to provide multiple instances of the same interface in a single component. A nesC component is either a *configuration* that contains a wiring of other components, or a *module* that contains an *implementation* of its interface methods. In Figure 1a, the `TimerC.Timer` interface is parameterized. The `Timer` interface of `SenseToInt` connects to a unique instance of the corresponding interface of `TimerC`. If another component connects to the `TimerC.Timer` interface, it will be connected to a different instance. Each timer can be initialized with different periods.

B. TinyOS

In TinyOS, there is a single thread of control managed by the scheduler, which may be interrupted by hardware events. Component methods encapsulate hardware interrupt handlers. Methods may transfer the flow of control to another component by calling a `uses` method. Computation performed in a sequence of method calls must be short, or it may block the processing of other events. A long running computation can be encapsulated in a *task*, which a method *posts* to the scheduler task queue. The TinyOS scheduler processes the tasks in the queue in FIFO order whenever it is not executing an interrupt handler. Tasks are atomic with respect to other tasks and do not preempt other tasks.

When a user compiles a TinyOS program for a sensor node, the nesC compiler automatically searches the TinyOS component library paths for included components, including directories containing the components that encapsulate the hardware components specific to the target platform, such as the clock, radio, and sensors. The nesC compiler generates a pre-processed C file, which can then be sent to a cross compiler for the target hardware.

C. TOSSIM

TinyOS programs can also be compiled for simulation on a PC. In this case, the nesC compiler follows the procedure just described but replaces the TinyOS scheduler and device drivers with TOSSIM code. The TOSSIM scheduler contains a task queue similar to the regular TinyOS scheduler. However, the TOSSIM scheduler also contains an ordered event queue. Events in this queue have a timestamp implemented as a `long long` in C (a 64-bit integer on most systems). The smallest time resolution is equal to 1 / 4MHz, the original CPU frequency of the Rene/Mica motes. Upon initialization, TOSSIM inserts a boot up event into the event queue. In the main scheduling loop, the TOSSIM scheduler begins by processing all tasks in the task queue in FIFO order. If there is an event in the event queue, it updates the simulated time with the timestamp of the new event and then processes the event. The processing of an event may cause tasks to be posted to the task queue and creation of new events with time stamps possibly equal to the current time stamp.

TOSSIM allows one or more nodes with the same TinyOS program to be simulated by maintaining a copy of the state of each component for each simulated node. Support for these copies is built into the nesC compiler so that the user does not need to modify the TinyOS program source code. TOSSIM has built-in models for radio connectivity between multiple nodes, and per-node ADC (analog-to-digital converter) values, as well as an interface for manually setting the per-node and per-link values and probabilities.

D. VisualSense

VisualSense is a modeling and simulation framework for wireless sensor networks that builds on Ptolemy II [5]. This framework supports actor-oriented definition of sensor nodes, wireless communication channels, physical media such as acoustic channels, and wired subsystems. The software architecture consists of a set of base classes for defining channels and sensor nodes, a library of subclasses that provide specific channel models and node models, and an extensible visualization framework. Custom nodes can be defined by subclassing the base classes and defining the behavior in Java or by creating composite models using any of several Ptolemy II modeling environments. Custom channels can be defined by subclassing the WirelessChannel base class and by attaching functionality defined in Ptolemy II models.

To support this style of modeling, VisualSense uses a specialization of the discrete-event (DE) domain of Ptolemy II. The DE domain of Ptolemy II [6] provides execution semantics where interaction between components occurs via events with time stamps. A sophisticated calendar-queue scheduler is used to efficiently process events in chronological order. The DE domain has a formal semantics that ensures determinate execution of deterministic models [7], although stochastic models for Monte Carlo simulation are also well supported. The precision in the semantics prevents the unexpected behavior that sometimes occurs due to modeling idiosyncrasies in some modeling frameworks.

The DE domain in Ptolemy II supports models with dynamically changing interconnection topologies. Changes in connectivity are treated as mutations of the model structure. The software is carefully architected to support multithreaded access to this mutation capability. Thus, one thread can be executing a simulation of the model while another changes the structure of the model, for example by adding, deleting, or moving actors, or changing the connectivity between actors. The results are predictable and consistent.

The most straightforward uses of the DE domain in Ptolemy II are similar to other discrete-event modeling frameworks such as NS, Opnet, and VHDL. Components (which are called *actors*) have *ports*, and the ports are interconnected to model the communication topology. Ptolemy II provides a visual editor for constructing DE models as block diagrams. VisualSense is a subclass of the DE modeling framework in Ptolemy II that is specifically intended to model sensor networks. In particular, it removes the need for explicit connections between ports, and instead associates ports with channels by name (e.g., "RadioChannel"). Connectivity can then be determined on the basis of the physical locations of the components. The algorithm for determining connectivity is itself encapsulated in a component as a channel model, and hence can be developed by the model builder. In VisualSense, sensor nodes themselves can be modeled in Java, or more interestingly, using more conventional DE models (as block diagrams) or other Ptolemy II models (such as dataflow models, finite-state machines or continuous-time models).

Ptolemy II also permits customized icons for components in a model. This can be used to render as part of a model useful visualizations that lend insight into the behavior of models. For example, a sensor node can have as an icon a translucent circle that represents (roughly or exactly) its transmit range.

Another feature of Ptolemy II is its sophisticated type system [8]. In this type system, actors, parameters, and ports can all impose constraints on types, and a type resolution algorithm identifies the most specific types that satisfy all the constraints. By default, the type system in Ptolemy II includes a type constraint for each connection in a block diagram. However, in wireless models, these connections do not represent all the type constraints. In particular, every actor that sends data to a wireless channel requires that every recipient from that channel be able to accept that data type. VisualSense imposes this constraint in the WirelessChannel base class, so unless a particular model builder needs more sophisticated constraints, the model builder does not need to specify particular data types in the model. They will be inferred from the ultimate sources of the data and propagated throughout the model.

III. Viptos

Viptos provides a bridge between VisualSense and nesC/TinyOS/TOSSIM by enabling the graphical development and interrupt-level simulation of actual TinyOS programs, with packet-level simulation of the network, while allowing the developer to use other models of computation available in Ptolemy II for modeling various parts of the system. We describe the architecture of this system in detail, including the representation of nesC components in Ptolemy II, the transformation of the TinyOS component library and existing TinyOS applications into this representation, generation of code for TinyOS programs developed in Viptos, and simulation of sensor network models that include nodes running TinyOS.

A. Representation of nesC components in Ptolemy II

In Ptolemy II, basic executable code blocks are called *actors* and may contain input and output *ports*. A port may be a simple port that allows only a single connection, or it may be a *multiport* that allows multiple connections. Fan-in or fan-out to simple ports may be achieved by placing a *relation* (the diamond-shaped icons in Figure 2) in the path of the connection. A code block is stored in a *class*, and an actor is an instance of the class.

We have developed the following representation scheme for the various parts of nesC components in Ptolemy II. We represent nesC components with Ptolemy II classes, and nesC component interfaces with Ptolemy II ports. We represent nesC `uses` interfaces with Ptolemy II output ports, and nesC `provides` interfaces with Ptolemy II input ports. We currently represent non-parameterized interfaces with simple ports; and single-index, parameterized interfaces with multiports.[1] Although multiple-index parameterized interfaces are allowed in nesC, Viptos does not support them, since they are not used in practice and do not appear in any existing components in the TinyOS component library. Figure 2c shows a graphical representation in Viptos of the equivalent wiring diagram for the `SenseToLeds` configuration shown in Figure 1a. Note that in Figure 2c, the `TimerC` component provides a parameterized interface, or input multiport, as indicated by the white triangle pointing into the block. Non-parameterized interfaces, or simple ports, are represented with the black triangles.

B. Transformation of the TinyOS component library and applications

Ptolemy II uses an XML-based language called MoML (Modeling Markup Language) [9] to specify interconnections of parameterized, hierarchical components. Viptos provides a tool called *nc2moml* for harvesting existing nesC files in the TinyOS 1.x component library and converting them into MoML for use within the Ptolemy II framework. We implemented the first version of the tool by modifying the nesC 1.1 compiler. The current version of *nc2moml* uses the XML output feature of the nesC 1.2 compiler. In both versions, the tool uses information about the component interfaces to generate MoML syntax that specifies the name of the component, as well as the name and input/output direction of each port, and whether it is a multiport. The resulting MoML files are used in Viptos to display TinyOS components as a library of graphical blocks. The user may drag and drop components from the library onto the workspace and create connections between component interfaces by clicking and dragging between ports. Figure 3 shows generated MoML code for the `TimerC` component referenced in Figure 1a. Figure 2c shows a TinyOS program created using components from the converted library.

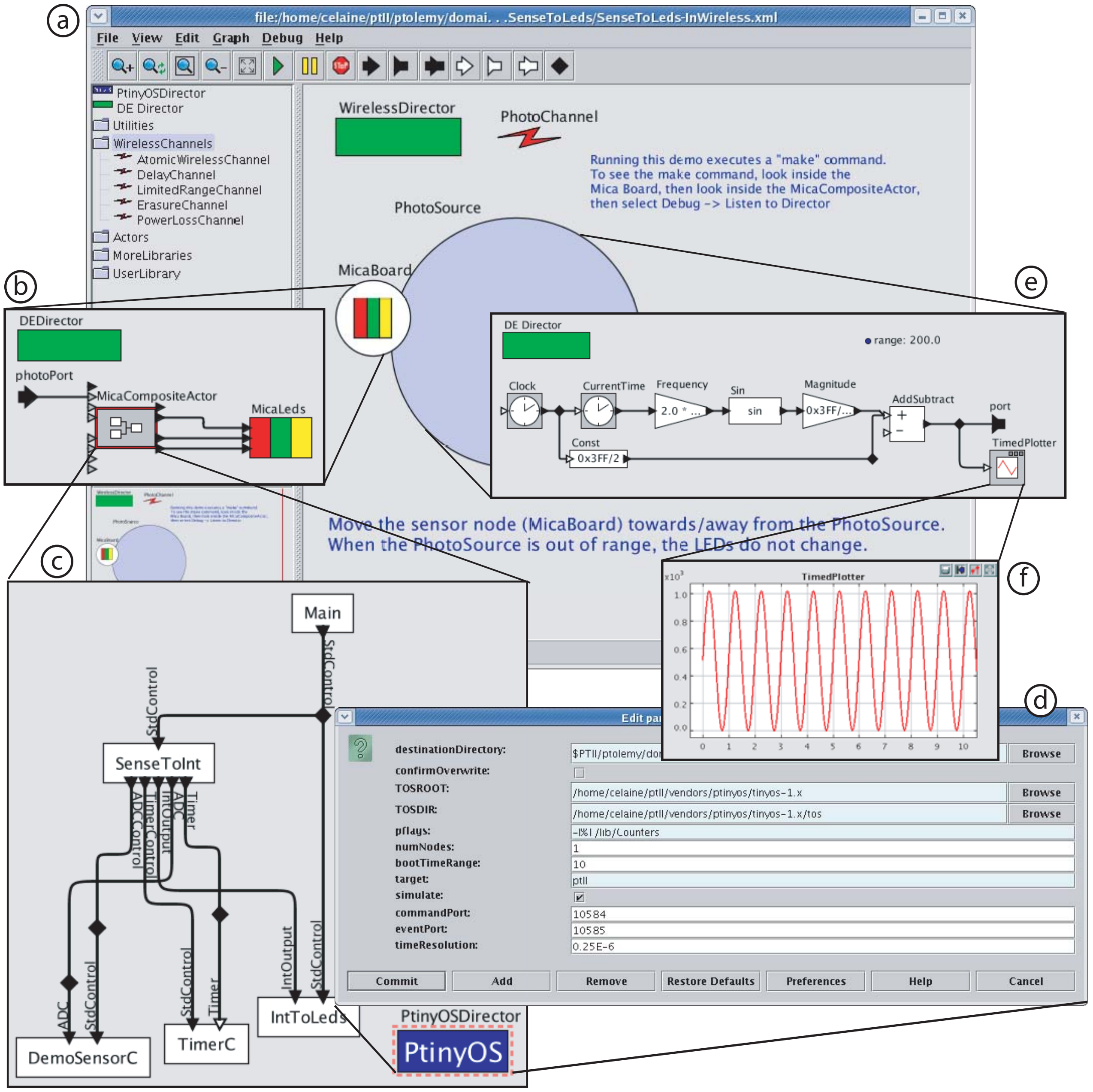
Viptos provides another harvest tool, *ncapp2moml*, which converts existing TinyOS 1.x application files into Viptos models. Whereas *nc2moml* only examines the interfaces contained in the nesC file, TinyOS application files in nesC do not have interfaces. *ncapp2moml* uses information about the nesC wiring graph and the referenced interfaces to generate MoML syntax that specifies a model that contains the actor corresponding to each nesC component used, the relations required at each port, and the links between the ports and relations such that the connections in the model correspond to the connections between interfaces in the nesC file. *ncapp2moml* can also automatically embed the converted TinyOS application into a template model containing a representation of the node. Figure 4 shows an example of a portion of the MoML code generated from the file shown in Figure 1a.

[1]See Section V for how we plan to change this to better support special cases where there are multiple connections to a single non-parameterized `provides` interface.

C. Generation of code for simulation and target deployment

Viptos can serve as both a program editing environment and a simulation environment. In both cases, an entity called the PtinyOS Director is placed into the workspace containing the nesC components. The PtinyOS Director controls code generation, compilation, simulation, and deployment to target hardware for a single node. Running the model shown in Figure 2c causes the PtinyOS Director to generate a nesC component file for `SenseToLeds`, equivalent to that shown in Figure 1a, as well as a makefile. The user can configure the PtinyOS Director (see Figure 2d) to compile the generated nesC code to any target supported by the TinyOS make system, including cross-compilation to target hardware, or TOSSIM for external simulation. The user can also load code to the target hardware from the Viptos interface.

To use Viptos as a simulation environment, the PtinyOS Director and the nesC components for the program graph should be embedded within a PtinyOSCompositeActor. The PtinyOSCompositeActor should be controlled by a DE Director. Figure 2b shows an example of this placement, where MicaCompositeActor is a subclass of PtinyOSCompositeActor that provides a representation of the Mica hardware interface, including ports for the ADC channels connected to sensors including a thermistor, photoresistor, microphone, magnetometer, and accelerometer; and also ports for the LEDs and radio communication. The user specifies the `ptII` simulation target as the target compilation platform. When the model is run, the PtinyOS Director generates a nesC file and a makefile, and compiles the nesC file against a custom version of TOSSIM to create a shared library.



Running this demo executes a "make" command. To see the make command, look inside the Mica Board, then look inside the MicaCompositeActor, then select Debug -> Listen to Director

Move the sensor node (MicaBoard) towards/away from the PhotoSource. When the PhotoSource is out of range, the LEDs do not change.

destinationDirectory:	$PTII/ptolemy/dor
confirmOverwrite:	☐
TOSROOT:	/home/celaine/ptII/vendors/ptinyos/tinyos-1.x
TOSDIR:	/home/celaine/ptII/vendors/ptinyos/tinyos-1.x/tos
pflags:	-I%I /lib/Counters
numNodes:	1
bootTimeRange:	10
target:	ptII
simulate:	✓
commandPort:	10584
eventPort:	10585
timeResolution:	0.25E-6

Fig. 2. SenseToLeds application in Viptos.

```
<?xml version="1.0"?>
<!DOCTYPE plot PUBLIC "-//UC Berkeley//DTD MoML 1//EN"
    "http://ptolemy.eecs.berkeley.edu/xml/dtd/MoML_1.dtd">

<class name="TimerC"
      extends="ptolemy.domains.ptinyos.lib.NCComponent">
  <property name="source"
      value="$CLASSPATH/tos/system/TimerC.nc" />
  <property name="_displayedName" class="..."
      value="TimerC" />
  <port name="StdControl" class="ptolemy.actor.IOPort">
    <property name="input" />
    <property name="_showName" class="..." />
  </port>
  <port name="Timer" class="ptolemy.actor.IOPort">
    <property name="input" />
    <property name="multiport" />
    <property name="_showName" class="..." />
  </port>
</class>
```

Fig. 3. Generated MoML for TimerC.nc

```
...
  <entity name="MicaCompositeActor"
     class="ptolemy.domains.ptinyos.lib.MicaCompositeActor">
    ...
    <entity name="DemoSensorC"
      class="tos.sensorboards.micasb.DemoSensorC" />
    <entity name="TimerC" class="tos.system.TimerC" />
    <entity name="Main" class="tos.system.Main" />
    <entity name="SenseToInt"
      class="tos.lib.Counters.SenseToInt" />
    <entity name="IntToLeds"
      class="tos.lib.Counters.IntToLeds" />
    <relation name="relation1"
      class="ptolemy.actor.IORelation" />
    <relation name="relation2"
      class="ptolemy.actor.IORelation" />
    <relation name="relation3"
      class="ptolemy.actor.IORelation" />
    <relation name="relation4"
      class="ptolemy.actor.IORelation" />
    <relation name="relation5"
      class="ptolemy.actor.IORelation" />
    ...
    <link relation="relation1" port="Main.StdControl"/>
    <link port="IntToLeds.StdControl" relation="relation2"/>
    <link relation1="relation2" relation2="relation1"/>
    <link port="SenseToInt.StdControl" relation="relation3"/>
    <link relation1="relation3" relation2="relation1"/>
    <link relation="relation4" port="SenseToInt.Timer"/>
    <link port="TimerC.Timer" relation="relation5"/>
    <link relation1="relation5" relation2="relation4"/>
    ...
  </entity>
...
```

Fig. 4. Generated MoML for SenseToLeds.nc

The PtinyOS Director also generates a Java wrapper to load the shared library into Viptos so that it can be run via JNI method calls.

D. Simulation of TinyOS in Viptos

In this section, we explain how Viptos simulates TinyOS programs from within the Ptolemy II framework.

1) Scheduling: Each instance of the PtinyOS Director compiles a custom copy of TOSSIM and uses it in its single node mode. This custom copy is a modified version of TOSSIM in which the scheduler and device driver functions contain additional JNI (Java Native Interface) calls to the PtinyOS Director with which it was compiled. JNI allows calls to be made between the Ptolemy II Java-based environment and the TOSSIM C-based environment.

All TOSSIM components call the `queue_insert_event()` function to insert new events into the TOSSIM event queue. The Viptos version of this function also makes a JNI call to insert equivalent events into the event queue of the DE Director controlling the PtinyOS Director using `fireAt()` with the TOSSIM system time as the argument. At each event timestamp, Viptos calls the modified TOSSIM scheduler to process the event. The modified TOSSIM scheduler updates the TOSSIM system time, processes an event in the TOSSIM event queue, and then processes all tasks in the task queue. If the TOSSIM event queue contains another event with the current TOSSIM system time, the scheduler processes the event along with any tasks that may have been generated. This last step is repeated until there are no other events with the current TOSSIM system time. Note that the order in the main loop is opposite that of the original TOSSIM, which processes all tasks before updating the TOSSIM system time and processing an event in the TOSSIM event queue. This change is required in order to guarantee causal execution in Viptos, since tasks may generate events with the current TOSSIM time stamp. Otherwise, new events may have a time stamp that is before the current Ptolemy II system time.

2) Type system: Communication between actors in Ptolemy II occurs through typed tokens. However, nesC components do not use Ptolemy II tokens; they use the C type system instead. To facilitate the embedding of a different type system within Ptolemy II, the PtinyOSCompositeActor subclasses the TypeOpaqueCompositeActor, which allows the actor's ports to have types, but does not require that the actors inside be typed. Thus, the inside of such an actor is not part of the Ptolemy II type system. In this case, the actors inside the PtinyOSCompositeActor are nesC components that use the C type system.

The PtinyOS Director and the modified TOSSIM automatically perform conversion between the token types used in Ptolemy II and the C types used in nesC. For example, most actors used in the DE domain of Ptolemy II communicate via tokens with values of type double. However, the ADC channel of a mote uses 10-bit unsigned values. When an ADC value is requested by TOSSIM, Viptos automatically performs the lossy conversion from a double-valued token in Ptolemy II to an unsigned short integer value in TOSSIM that is masked for 10-bit usage. In another example, when TOSSIM updates the state of the LEDs, Viptos automatically converts the `char` representing the LED value in TOSSIM into a boolean-valued token in Ptolemy II.

3) I/O: In the DE domain of Ptolemy II, tokens received at the input port of an actor will cause the actor to fire at the time of the token timestamp. The token is usually consumed, at which point the port is empty. The PtinyOSCompositeActor may receive tokens on the ADC ports that represent new values. To reconcile the difference in timing between when the simulated environment makes a new ADC value available and when the simulated node reads its ADC ports, we use a Ptolemy II PortParameter instead of a Port for the ADC ports. Our usage of the PortParameter makes the port value persistent between updates such that when the TinyOS program requests data from the ADC port, it gets the value of the most recently received token.

The DE model containing the PtinyOS Director and nesC components can be embedded in a VisualSense Wireless model so that the physical environment and radio channels can be simulated. Figure 2a shows an example of embedding a node running the `SenseToLeds` TinyOS program in the Wireless domain. Viptos overrides the built-in ADC and radio models and LED device drivers in the original version of TOSSIM so that they can send data to and receive data from the ports of the PtinyOSCompositeActor. This allows the simulated node to interact with user-created models of sources of light (see Figures 2e and 2f), temperature, radio channels, other nodes, etc.

4) Multiple nodes: By embedding multiple PtinyOSCompositeActors, each controlled by a different PtinyOS Director, into the Wireless domain, multiple nodes with different programs can be simulated at the same time. Separately compiled and loaded shared libraries prevent namespace collision between different simulated TinyOS programs.

Figure 5 shows an example model that contains two nodes that communicate over a lossless radio channel. The first node contains the `CntToLedsAndRfm` TinyOS program, which maintains an increasing integer counter and broadcasts the value over the radio and displays the value in binary on the LEDs. The second node contains the `RfmToLeds` TinyOS program, which receives the counter value over the radio and displays the value in binary on the LEDs. The radio channel model can easily be replaced by deleting it and dragging in a different channel model from the menu in the left-hand pane.

IV. Related Work

A. Modeling and simulation environments for wireless systems

A number of frameworks for modeling wireless systems are available.

ns-2 is a well-established, open-source network simulator. It is a discrete event simulator with extensive support for simulating TCP/IP, routing, and multicast protocols over wired and wireless (local and satellite) networks. The wireless and mobility support in ns-2 comes from the Monarch project, which provides channel models and wireless network layer components in the physical, link, and routing layers.

SensorSim [10] also builds on ns-2 and claims power models and sensor channel models. A power model consists of an energy provider (the battery) and a set of energy consumers (CPU, radio, and sensors). An energy consumer can have several modes, each corresponding to a different trade-off between performance and power. The sensor channels model the dynamic interaction between the physical environment and the sensor nodes. SensorSim also claims hybrid simulation in which real sensor nodes can participate. Unfortunately, SensorSim is no longer under development and will not be publicly released.

OPNET Modeler is a commercial tool that offers sophisticated modeling and simulation of communication networks. An OPNET model is hierarchical, where the top level contains the communication nodes and the topology of the network. Each node can be constructed from software components, called processes, in a block-diagram fashion, and each process can be constructed using finite state machine (FSM) models. It uses a discrete event simulator to execute the entire model. In conventional OPNET models, nodes are connected by static links. The OPNET Wireless Module provides support for wireless and mobile communications. It uses a 13-stage "transceiver pipeline" to dynamically determine the connectivity and propagation effects among nodes. Users can specify transceiver frequency, bandwidth, power, and other characteristics. These characteristics are used by the transceiver pipeline stages to calculate the average power level of the received signals to determine whether the receiver can receive this signal. In addition, antenna gain patterns and terrain models are well supported.

OMNET++ [11] is an open source tool for discrete-event modeling. With the Mobility Framework extension, it shares many concepts, solutions and features with OPNET. But instead of using FSM models for processes, it defines a component interface for the basic module, with a set of methods including initialize(), handleMessage(), and finish(), that are overridden by the model builder. The initialize() method is called by the simulator at the beginning of executing the network, the handleMessage() method is called when a message is sent to this module, and the finish() method is called when the execution is prepared to stop. This object-oriented approach is similar to the abstract semantics of Ptolemy II [5]. The NesCT tool of the EYES WSN project allows users to run TinyOS applications directly in OMNeT++ simulations. However, according to the documentation, simulated nodes can only send radio messages; they cannot receive radio messages.

J-Sim [12] is an open-source, component-based, compositional network simulation environment that is developed entirely in Java. A new wireless sensor framework [13] is being developed that builds upon the autonomous component architecture (ACA) and the extensible internetworking framework (INET) of J-Sim, and provides an object-oriented definition of (i) target, sensor and sink nodes, (ii) sensor and wireless communication channels, and (iii) physical media such as seismic channels, mobility model and power model (both energy-producing and energy-consuming components). Application-specific models can be defined by sub-classing classes in the simulation framework and customizing their behaviors. It also includes a set of classes and mechanisms to realize network emulation. This new framework extends the notion of network emulation to Berkeley Mica mote-based WSNs, which are used to extract physical environment data by using SerialForwarder, a utility distributed with TinyOS that collects TinyOS packets sent to a mote base station attached to a PC and forwards them through the serial port.

Prowler [14] is a probabilistic wireless network simulator running under MATLAB capable of simulating wireless distributed systems, from the application to the physical communication layer. Although Prowler provides a generic simulation environment, its current target platform is the Berkeley Mica mote running TinyOS. Prowler is an event-driven simulator that can be set to operate in either deterministic mode (to produce replicable results while testing the application) or in probabilistic mode that simulates the nondeterministic nature of the communication channel and the low-level communication protocol of the motes. It can incorporate arbitrary number of motes, on arbitrary (possibly dynamic) topology, and it was designed so that it can easily be embedded into optimization algorithms.

Em* [15] is toolsuite for developing sensor network applications on Linux-based hardware platforms called microservers. It supports deployment, simulation, emulation, and visualization of live systems, both real and simulated. EmTOS [16] is an extension to Em* that enables an entire nesC/TinyOS application to run as a single module in an Em* system. The EmTOS wrapper library is similar to the TOSSIM simulated device library. Em* modules are implemented as user-space processes that communicate through message passing via device files. This means that the minimum granularity of a timer is 10ms, corresponding to the Linux jiffy clock that is part of the scheduler in the Linux 2.4 kernel. Thus, EmTOS modules are restricted to using the Linux scheduler as the main programming model.

TinyViz [3] is a Java-based graphical user interface for TOSSIM. TinyViz supports software plugins that watch for events coming from the simulation – such as debug messages, radio messages, and so forth – and react by drawing information on the display, setting simulation parameters, or actuating the simulation itself, for example, by setting the sensor values that simulated motes will read. TinyViz includes a radio model plugin with two built-in models: "Empirical" (based on an outdoor trace of packet connectivity with the RFM1000 radios) and "Fixed radius" (all motes within a given fixed distance of each other have perfect connectivity, and no connectivity to other motes).

Other simulators used in the TinyOS community for cycle accurate simulation/emulation of the Atmel AVR (processor used in the Mica mote series) instruction set include ATEMU [17] and Avrora [18]. ATEMU simulates a byte-oriented interface to the radio and its transmissions at the bit level with precise timing. Avrora works at the byte level with precise timing, and its simulation speed scales much better than ATEMU for large number of nodes. Both support simulation of heterogeneous networks.

All of these systems provide extension points where model-builders can define functionality by adding code. Some are also open-source software, like Viptos. All except EmStar provide some form of discrete-event simulation, but none provide the ability that Viptos inherits from Ptolemy II to integrate diverse models of computation, such as continuous-time, dataflow, synchronous/reactive, and time-triggered. This capability can be used, for example, to model the physical environment, as well as the physical dynamics of mobility of sensor nodes, their digital circuits, energy consumption and production, signal processing, or real-time software behavior. Such models would have to be built with low-level code. Ptolemy II supports hierarchical nesting of heterogeneous models of computation [5]. It also appears to be unique among these modeling environments in that FSM models can be arbitrarily nested with other models; i.e., they are not restricted to be leaf nodes [19]. It also appears to be the only one to provide a modern type system at the actor level (vs. the code level) [8].

B. *TinyOS development and editing environments*

GRATIS II (Graphical Development Environment for TinyOS) is built on top of GME 3 (Generic Modeling Environment). The TinyOS component library is available as graphical blocks with GRATIS II. Given a valid model, the GRATIS II code generator can transform all the interface and wiring information into a set of nesC target files. However, GRATIS II was developed mainly for static analysis of TinyOS component graphs and does not support simulation.

TinyDT is a TinyOS 1.x plugin for the Eclipse platform that implements an IDE (integrated development environment) for TinyOS/nesC development. This open source project features syntax highlighting of nesC code, code navigation, code completion for interface members, support for multiple target platforms and sensor boards, automatic build support, team development support (through Eclipse-CVS integration), and support for multiple TinyOS source trees. TinyDT uses a Java nesC parser implemented using ANTLR to build an in-memory representation of the actual nesC application, which includes component hierarchy, wirings, interfaces and the JavaDoc style nesC documentation. TinyOS IDE is another Eclipse plugin that supports TinyOS project development and provides nesC syntax highlighting. Both TinyDT and TinyOS IDE complement Viptos in that they can be used to create and edit the source code for new TinyOS library components, which can then be imported into Viptos for simulation using *nc2moml*.

V. Work in Progress

This section describes features of Viptos that are currently being changed as we further develop the framework.

A. *Ports*

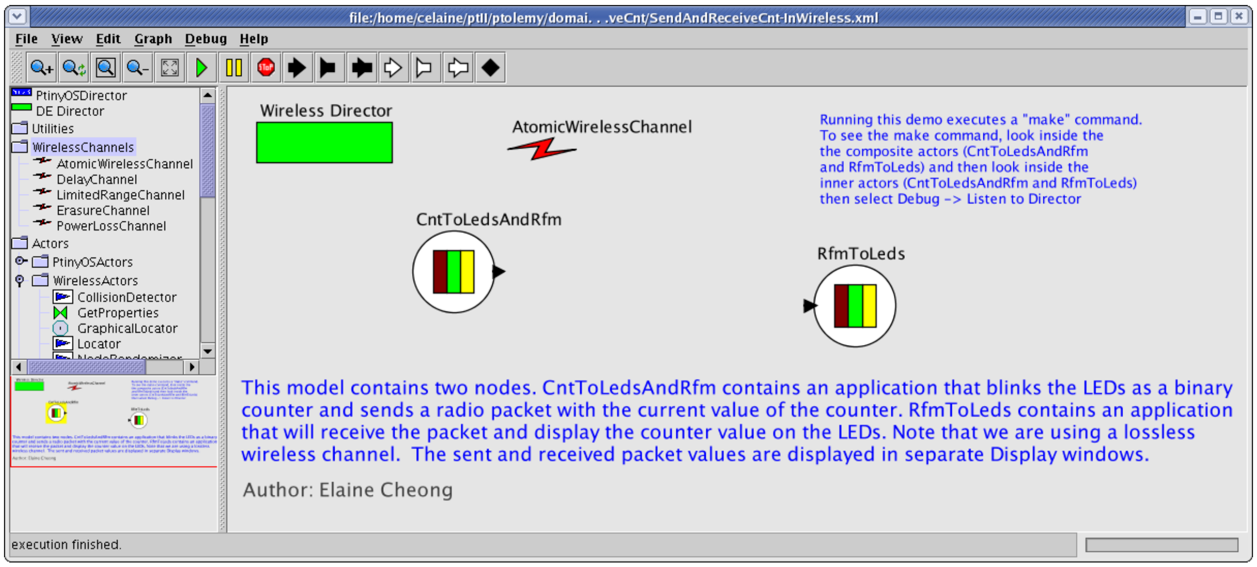
The current mapping of non-parameterized nesC interfaces to Ptolemy II simple ports and parameterized nesC interfaces to Ptolemy II multiports leads to an inability to express certain types of nesC configurations. For example, if a configuration contains the following mapping, where `a`, `b`, `c`, and `d` are non-parameterized interfaces:

```
a -> d
b -> d
a -> c
```

Then, the original Viptos mapping will produce an extra connection between `b` and `c`, since in Ptolemy II, relations are required to create multiple connections to port `d`, and relations that are connected to each other are considered to be part of a relation group in which the relations are indistinguishable from each other, and connections between relations are directionless.

Similarly, multiple connections between the same `uses` and `provides` interfaces may be lost or lead to extra connections when translating from nesC to MoML. Since relations in a group are indistinguishable from each other, multiple connections between relations cannot be represented in Ptolemy II.

We plan to change the current multiport/simple port distinction and represent both parameterized and non-parameterized nesC interfaces



Running this demo executes a "make" command.
To see the make command, look inside the
the composite actors (CntToLedsAndRfm
and RfmToLeds) and then look inside the
inner actors (CntToLedsAndRfm and RfmToLeds)
then select Debug -> Listen to Director

This model contains two nodes. CntToLedsAndRfm contains an application that blinks the LEDs as a binary counter and sends a radio packet with the current value of the counter. RfmToLeds contains an application that will receive the packet and display the counter value on the LEDs. Note that we are using a lossless wireless channel. The sent and received packet values are displayed in separate Display windows.

Author: Elaine Cheong

execution finished.

Fig. 5. SendAndReceive application in Viptos.

with multiports. We plan to attach a Ptolemy II parameter to multiports that represent parameterized nesC interfaces. The value of the parameter will be an array of integers that is constrained to have a length equal to the number of connections made to the port. Using multiports for all connections will allow all types of connections that can be made in nesC. Note that multiple connections to the same `provides` port may actually be a sign of a possible race condition, since the provided code can be triggered by simultaneous events from the physical world. However, to avoid duplicate functionality, we rely on the nesC compiler to do a complete analysis of the connected interface methods to detect incorrect usage of commands or events marked with the `async` keyword and hence possible race conditions.

B. Multihop routing

We are currently working on supporting multihop routing within Viptos. Planned features include: setting node IDs external to TOSSIM, graphical support for visualizing the routing tree, and creating a special channel model that can parse TinyOS packets and direct them to the appropriate node.

VI. Conclusion and Future Work

We have described an extensible software framework for sensor network modeling called Viptos that is built on VisualSense and TinyOS. An important area of future research involves investigating the scalability of this framework as more nodes are simulated. One option is to investigate the scalability of building on top of the Avrora simulator instead of TOSSIM, since Avrora supports heterogeneous networks. However, this would make the simulation platform specific to the AVR processor family. Other interesting topics include how to enable code dissemination algorithms such as Deluge [20] from within the framework. We are also investigating how to represent the individual methods of an interface using ports or an alternate visual syntax.

Viptos is open-source software, freely available at http://ptolemy.eecs.berkeley.edu/viptos. We hope that the community can use this framework to encapsulate and exchange methods and expertise in channel modeling, sensor node design, and application development.

Acknowledgments

This work was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF award #CCR-0225610), the State of California Micro Program, and the following companies: Agilent, DGIST, General Motors, Hewlett Packard, Infineon, Microsoft, and Toyota.

The authors would like to thank Christopher Brooks for managing the Viptos code release, David Gay for creating the nesC 1.2 XML output feature and associated Java tools, and Jie Liu for providing helpful comments.

References

[1] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors," in *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM Press, 2000, pp. 93–104.

[2] 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) 2003*, June 2003.

[3] P. Levis, N. Lee, M. Welsh, and D. Culler, "TOSSIM: accurate and scalable simulation of entire tinyos applications," in *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems (SenSys 2003)*. ACM Press, 2003, pp. 126–137.

[4] P. Baldwin, S. Kohli, E. A. Lee, X. Liu, and Y. Zhao, "Modeling of sensor nets in Ptolemy II," in *IPSN'04: Proceedings of the Third International Aymposium on Information Processing in Sensor Networks*. New York, NY, USA: ACM Press, 2004, pp. 359–368.

[5] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming heterogeneity – the Ptolemy approach," *Proceedings of the IEEE, Special Issue on Modeling and Design of Embedded Software*, vol. 91, no. 1, pp. 127–144, January 2003.

[6] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Z. (eds.), "Heterogeneous concurrent modeling and design in Java: Volume 3: Ptolemy II domains," University of California, Berkeley, Tech. Rep. Technical Memorandum UCB/ERL M05/23, July 15 2005.

[7] E. A. Lee, "Modeling concurrent real-time processes using discrete events," *Ann. Softw. Eng.*, vol. 7, no. 1-4, pp. 25–45, 1999.

[8] Y. Xiong, "An extensible type system for component-based design," EECS Department, University of California, Berkeley, Tech. Rep. UCB/ERL M02/13, 2002.

[9] E. A. Lee and S. Neuendorffer, "MoML – a modeling markup language in XML – version 0.4," University of California at Berkeley, Tech. Rep., March 2000.

[10] S. Park, A. Savvides, and M. B. Srivastava, "SensorSim: a simulation framework for sensor networks," in *MSWIM '00: Proceedings of the 3rd ACM International Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems*. New York, NY, USA: ACM Press, 2000, pp. 104–111, see also http://nesl.ee.ucla.edu/projects/sensorsim/.

[11] A. Varga, "The OMNeT++ discrete event simulation system," in *Proceedings of the European Simulation Multiconference (ESM'2001)*, Prague, Czech Republic, June 6-9 2001, http://www.omnetpp.org/.

[12] H.-Y. Tyan, "Design, realization and evaluation of a component-based compositional software architecture for network simulation," Ph.D. dissertation, 2002, see also http://www.j-sim.org.

[13] A. Sobeih, W.-P. Chen, J. C. Hou, L.-C. Kung, N. Li, H. Lim, H.-Y. Tyan, and H. Zhang, "J-Sim: A simulation and emulation environment for wireless sensor networks," online, 2005, http://www.j-sim.org/v1.3/sensor/JSim.pdf.

[14] G. Simon, P. Völgyesi, M. Maróti, and Ákos Lédeczi, "Simulation-based optimization of communication protocols for large-scale wireless sensor networks," in *Proceedings 2003 IEEE Aerospace Conference*, vol. 3, 2003, pp. 3_1339–3_1346, see also http://www.isis.vanderbilt.edu/projects/nest/prowler/.

[15] L. Girod, J. Elson, A. Cerpa, T. Stathopoulos, N. Ramanathan, and D. Estrin, "EmStar: A software environment for developing and deploying wireless sensor networks," in *USENIX 2004 Annual Technical Conference*, 2004, pp. 283–296.

[16] L. Girod, T. Stathopoulos, N. Ramanathan, J. Elson, D. Estrin, E. Osterweil, and T. Schoellhammer, "A system for simulation, emulation, and deployment of heterogeneous sensor networks," in *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*. ACM Press, 2004, pp. 201–213.

[17] J. Polley, D. Blazakis, J. McGee, D. Rusk, J. S. Baras, and M. Karir, "Atemu: A fine-grained sensor network simulator," in *Proceedings of the First IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks (SECON'04)*, 2004.

[18] B. Titzer, D. Lee, and J. Palsberg, "Avrora: Scalable sensor network simulation with precise timing," in *Proceedings of IPSN'05, Fourth International Conference on Information Processing in Sensor Networks*, 2005, pp. 477–482.

[19] 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*, vol. 18, no. 6, June 1999.

[20] J. W. Hui and D. Culler, "The dynamic behavior of a data dissemination protocol for network programming at scale," in *Proceedings of the 2nd international conference on Embedded networked sensor systems*. ACM Press, 2004, pp. 81–94.