In Proc. of *Information Processing in Sensor Networks*, (IPSN),
April 26-27, 2004, Berkeley, CA, USA

Modeling of Sensor Nets in Ptolemy II

Philip Baldwin
Department of ECE
2 Mitchell
Charlottesville, VA 22904
pjb2e@virginia.edu

Sanjeev Kohli
Department of EECS
University of California, Berkeley
Berkeley, CA 94720, USA
sanjeev@eecs.berkeley.edu

Edward A. Lee
Department of EECS
University of California, Berkeley
Berkeley, CA 94720, USA
eal@eecs.berkeley.edu

Xiaojun Liu
Department of EECS
University of California, Berkeley
Berkeley, CA 94720, USA
liuxj@eecs.berkeley.edu

Yang Zhao
Department of EECS
University of California, Berkeley
Berkeley, CA 94720, USA
ellen_zh@eecs.berkeley.edu

ABSTRACT

This paper describes a modeling and simulation framework called *VisualSense* for wireless sensor networks that builds on and leverages Ptolemy II. 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 certain 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.

Categories and Subject Descriptors

I.6.7 [**Computing Methodologies**]: Simulation And Modeling – *Simulation Support Systems*

General Terms

Design, Experimentation, Verification.

Keywords

Simulation, Modeling, Wireless Networks, Discrete-Event Models, Ptolemy II, VisualSense.

1. INTRODUCTION

This paper describes a software package for modeling sensor networks that builds on and leverages Ptolemy II [5]. *VisualSense* is a *configuration* of Ptolemy II that provides component-based design of sensor networks and natural simulation and visualization. Such networks are highly dynamic, in that nodes come and go and their connectivity changes. Also, we need to combine multiple communication media (acoustic, optical, radio). In this paper, we describe a specialization of the discrete-event domain of Ptolemy II supporting this style of modeling.

The discrete-event (DE) domain of Ptolemy II [10] provides execution semantics where interaction between components is via events with time stamps. The time stamps are double-precision floating point numbers, and a sophisticated calendar-queue scheduler is used to efficiently process events in chronological order. DE has a formal semantics that ensures determinate execution of deterministic models [12], 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, as for example shown in Figure 1. However, such block diagrams are a poor representation of a sensor network, because the interconnection topology is highly variable.

This paper describes a subclass of the DE modeling framework in Ptolemy II that is specifically intended to model sensor networks. It largely preserves DE semantics, but changes the mechanism for connecting components. 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.

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). For example, a sensor node with modal behavior can be defined by sketching a finite-state machine and providing refinements to each of the states to define the behavior of the node in that state. This can be used, for example, to model energy consumption as a function of state, enabling potential representation of sensor nodes using *resource interfaces* [3]. Sophisticated models of the coupling between energy consumption and media access control protocols become possible.

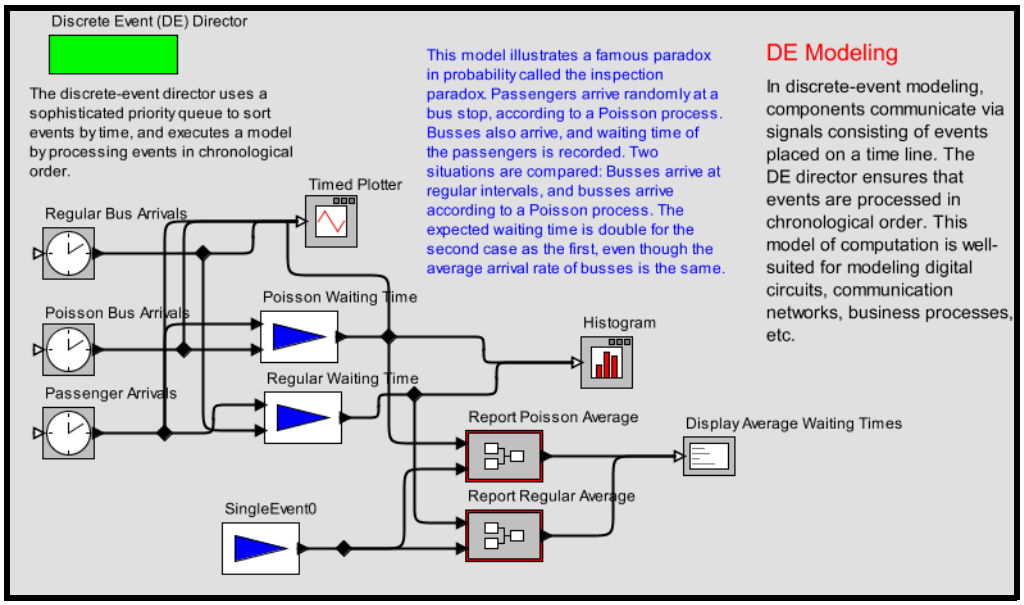

Figure 1. Typical DE model in Ptolemy, represented as a block diagram. Such block diagram representations are not well-suited to sensor network modeling.

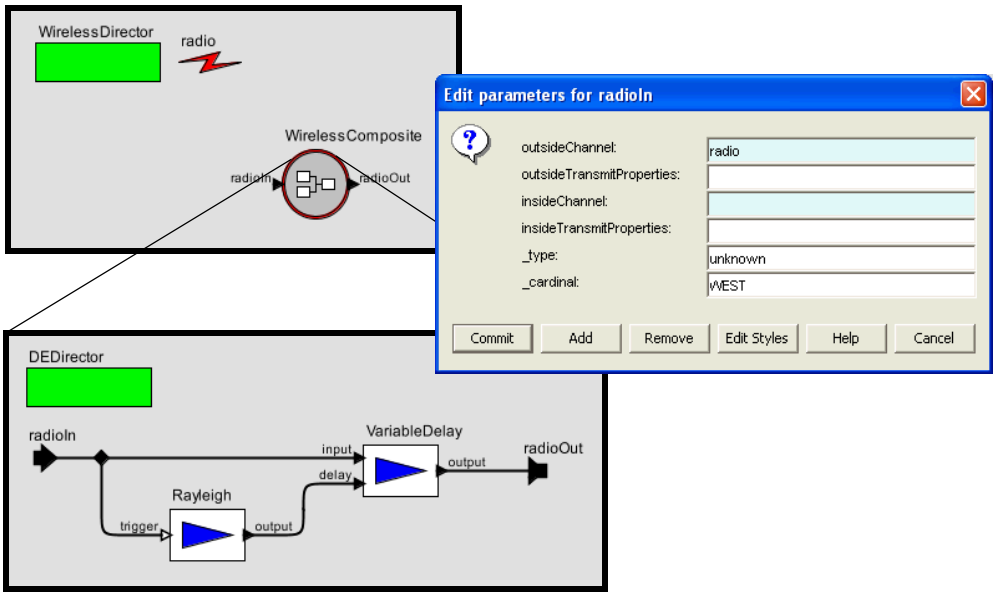

Figure 2. Example of a sensor node defined as a composite actor using a block diagram in the Ptolemy II discrete-event domain.

1.1 Related Work

A number of frameworks for modeling wireless systems are available.

Ns-2[15] is a well-established, open-source network simulator with many contributors at several institutions including LBL, Xerox PARC, UCB, and USC/ISI. 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 at CMU [14], which provides channel models and wireless network layer components in the physical, link, and routing layers. It provides a radio propagation model based on the two ray ground reflection approximation and a shared media model in the physical layer; it implements the IEEE802.11 MAC protocol in the link layer; it implements the dynamic source routing protocol, etc. in the routing layer.

SensorSim[17], from UCLA, 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. For example, the radio may have a number of transmit modes using different symbol rates and transmit power. The sensor channels model the dynamic interaction between the physi-

cal environment and the sensor nodes. SensorSim also claims hybrid simulation in which real sensor nodes can participate. Unfortunately, at the time of this writing, SensorSim has not been publicly released and does not appear to be under further development.

EmStar [6], a more recent project from UCLA, focuses on the problem of developing software for wireless sensor networks. Its execution environment supports simulation, deployment, and a hybrid mode that combines simulation with real wireless communication among sensors situated in the environment. Services provided or planned by EmStar include networking and time synchronization. In the programming model of EmStar, modules run within individual Linux processes that communicate through message passing via device files.

OPNET Modeler [16] is a commercial tool from OPNET Technologies, Inc., which has been engaged in network software and applications since 1986. It 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++[19] is an open source tool for discrete-event modeling. With the Mobility Framework extension[13], 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].

J-Sim [18] is built on object-oriented component principles. INET, a generalized packet switched network model, is implemented in J-Sim, together with a suite of Internet best effort, integrated services, and differentiated services protocols.

All of these systems provide extension points where model-builders can define functionality by adding code. Some are also open-source software, like VisualSense. All except EmStar provide some form of discrete-event simulation, but none provide the ability that VisualSense 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 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 [8]. It also appears to be the only one to provide a modern type system at the actor level (vs. the code level) [20] and to type check communication packet structure (using record types).

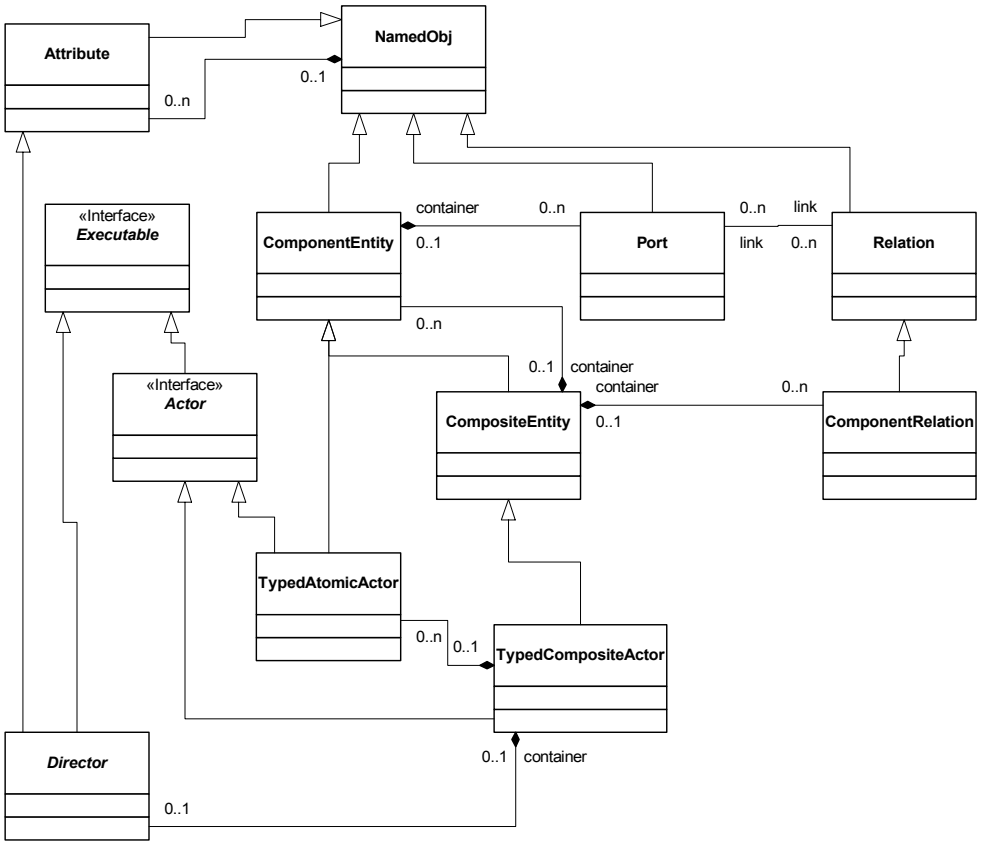

Figure 3. UML class diagram showing key classes in Ptolemy II, on which VisualSense is built.

1.2 Paper Overview

Section 2 describes how to build models. Section 3 gives the software architecture of VisualSense, with emphasis on customization mechanisms. Examples of models using the framework are described in Section 6.

2. BUILDING MODELS

VisualSense is constructed by subclassing key classes in Ptolemy II. The extension to Ptolemy consists of a few new Java classes and some XML files. The classes are designed to be subclassed by model builders for customization, although non-trivial models can also be constructed without writing any Java code. In the latter case, sensor network nodes are specified using block diagrams and finite state machines.

An example of a sensor node defined as a composite actor is shown in Figure 2. At the top of the figure is a segment of a wireless model showing a "WirelessDirector" (which specifies that the diagram is a wireless model), a channel definition named "radio" (with the lightning bolt icon), and a sensor node named "WirelessComposite." The sensor node has two ports named "radioIn" and "radioOut." The parameter screen for the radioIn port is shown at the right, where you can see that the *outsideChannel* parameter is set to "radio," which indicates that communication is via the channel named "radio."

The composite contains the model shown at the bottom of Figure 2. This model gives the behavior of the sensor node. It is not a wireless model, but is rather an ordinary discrete-event Ptolemy II model, like that of Figure 1, where communication is represented explicitly by lines in the block diagram. That it is a discrete-event model is specified by the "DEDirector" component. But a different model of computation could be used at this level as well. The parameter screen at the right in the figure shows that nothing is given for an *insideChannel*, which is another indication that at this lower level of the hierarchy, communication is not wireless.

The behavior of the sensor node example in Figure 2 is readily understood from the diagram. When a radio signal is received, it generates a random number according to a Rayleigh distribution, and then retransmits the same radio signal after a time delay given by that random number. Although this rather simplistic behavior does not constitute a useful sensor node, it is easy to imagine significant elaborations that leverage the considerable modeling expressiveness of Ptolemy II to describe much more complex behavior.

3. SOFTWARE ARCHITECTURE

The key classes in Ptolemy II (which define its meta model) are shown in Figure 3. Executable components implement the *Actor* interface, and can be either *atomic* or *composite*. Atomic actors are defined in Java, while composite actors are assemblies of actors and relations. Each actor, whether atomic or not, contains ports, which are linked in a composite actor via relations. A top-level model is itself a composite actor, typically with no ports. Actors, ports and relations can all have attributes (parameters). One of the attributes is a *director*. The director plays a key role in Ptolemy II: it defines the semantics of a composite. It gives the concurrency model and the communication semantics. In VisualSense, the director implements the simulator. The WirelessDirector is an almost completely unmodified subclass of the pre-existing discrete-event director (DEDirector) in Ptolemy II.

The extensions that constitute VisualSense are shown in Figure 4. A node in a wireless network is an actor that can be a subclass of either TypedAtomicActor or TypedCompositeActor. The difference between these is that for TypedAtomicActor, the behavior is defined in Java code, whereas for TypedCompositeActor, the behavior is defined by another Ptolemy II model, which is itself a composite of actors.

Actors that communicate wirelessly have ports that are instances of WirelessIOPort. As with any Ptolemy II port, the actor sends data by calling the *send* or *broadcast* method on the port. The *send* method permits specification of a numerically indexed subchannel, whereas the *broadcast* method will send to all subchannels.

In the case of WirelessIOPort, *send* and *broadcast* cannot determine the destination ports using block-diagram-style connectivity because there is no such connectivity. Instead, they identify an instance of WirelessChannel by name, and delegate to that instance to determine the destination(s) of the messages. The instance is specified by setting the *outsideChannel* parameter of the port equal to the name of the wireless channel (all actors at a given level of the hierarchy have unique names, a feature provided by the base class).

The WirelessChannel interface and the AtomicWirelessChannel base class, shown in Figure 4, are designed for extensibility. They work together with WirelessIOPort, which uses the public method, *transmit*, to send data. That method takes three arguments, a *token*[1] to transmit, a source port, and a token representing transmit properties (transmit power, for example, as discussed below).

AtomicWirelessChannel has a suite of protected methods, indicated in the UML diagram by the leading pound sign (#); in the Ptolemy II coding style, protected methods have names that begin with leading underscores(_). These provide simple default behavior, but are intended to be overridden in subclasses to provide more sophisticated channel models. This is an example of the *strategy design pattern* [7], where the code providing the large-scale behavior delegates to protected methods for detailed behavior.

The default behavior of AtomicWirelessChannel is represented by the following pseudo code:

```
public void transmit(token, sender, properties) {
   foreach receiver in range {
      _transmitTo(token, sender, receiver, properties)
   }
}
```

To determine which receivers are in range, it calls the protected method `_receiversInRange()`, which by default returns all receivers contained by ports that refer to the same channel name as that specified by the sender. The `_transmitTo()` method by default uses the public `transformProperties()` method to modify the properties argument (see below) and then put the token and the modified properties into the receiver. The `transformProperties()` method

[1] A token in Ptolemy II is a wrapper for data. Ptolemy II provides a rich set of data types encapsulated as tokens, including composite types such as arrays, matrices, and records (which have named fields). A sophisticated type system ensures validity of data that is exchanged via tokens. A rich expression language permits definition of tokens as expressions that can depend on parameters of actors or ports. Scoping rules limit the visibility of parameters according to the hierarchy of the model, thus avoiding the pitfalls of using global variables. For details, see [9].

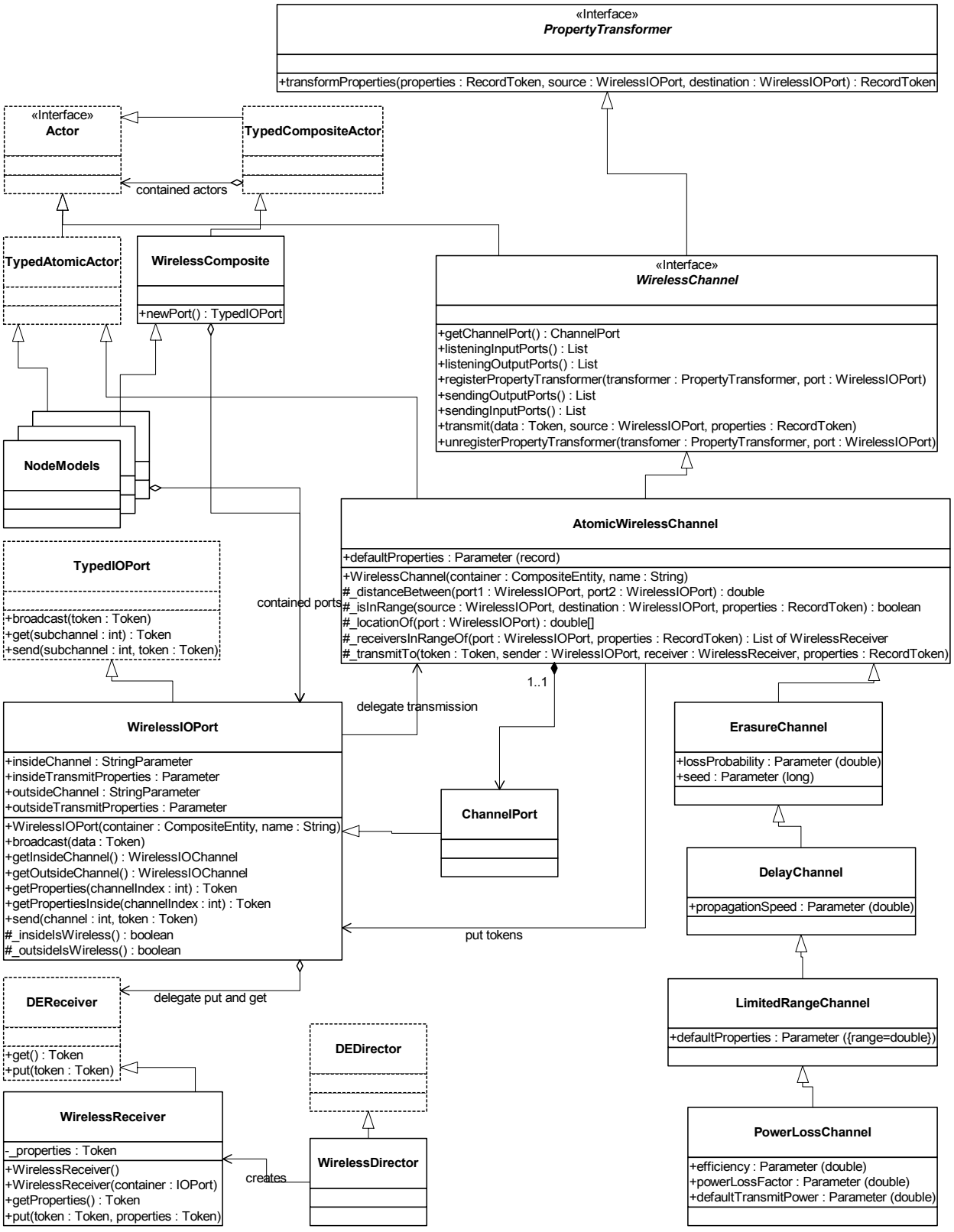

Figure 4. UML class diagram showing the key classes for wireless sensor net modeling. These classes plus some XML files specifying configuration information and libraries constitute VisualSense.

applies any property transformers that have registered using the `registerPropertyTransformer()` method, but does nothing further. Thus, if there are no registered properties transformers, the default AtomicWirelessChannel has no range limitations and introduces no transmission degradations. We can now show through a series of examples how subclassing makes it easy to construct more detailed (and useful) channel models.

4. MODEL COMPONENTS

We illustrate the contruction of model components such as channel models with examples.

4.1 Erasure Channel

Consider a channel that randomly drops data. This can be defined as follows:

```
public class ErasureChannel extends AtomicWirelessChannel{
   ... specify constructor ...
   public Parameter lossProbability;
   public Parameter seed;
   private Random _random = new Random();
   public void transmit(token, sender, properties) {
      double experiment = _random.nextDouble();
      if (experiment >= lossProbability.doubleValue()) {
         super.transmit(token, sender, properties);
      }
   }
}
```

It is that simple. This channel adds to the base class a parameter called *lossProbability*. (The details of constructing the channel and this parameter are not shown, see [9]). The Java class Random is used to "throw the dice" to determine whether or not the transmission should actually occur.

Note that the above channel model might not be exactly what you want. In particular, it throws the dice once, and uses the result to decide whether or not to transmit to all recipient ports that are in range. A better design might throw the dice once for each recipient port. We leave it as a (simple) exercise for the reader to see how to modify the above code to accomplish this.

4.2 Limited Range Channels

The above channels have unlimited range, in that any input port that references the channel by name is in range. This is because the default implementation of `_isInRange()` simply returns true. It is easy for a subclass to change this behavior. Consider, for example, a channel model that uses a distance threshold:

```
public class LimitedRangeChannel extends ErasureChannel {
   ... specify constructor ...
   public Parameter range;
   protected boolean _isInRange(
         source, destination, properties) {
      double distance = _distanceBetween(source, destination);
      if (distance <= range.doubleValue()) {
         return true;
      } else {
         return false;
      }
   }
}
```

This class overrides the `_isInRange()` method to simply check the distance between the source and the destination, returning true if the distance is below the specified *range* threshold. The `_isInRange()` method uses the `_locationOf()` method, which by default returns the (two-dimensional) location of the icon within the visual renditions of the model. Again, this yields a simplistic model, but nonetheless one that could be useful. It would be easy to build a variant where location is in three dimensional space and is specified by attributes attached to the sensor nodes. More sophisticated models of range rely on the *transmit properties* concept, which we explain next.

4.3 Transmit Properties

In the previous section, the range of wireless communication is a property of the channel. However, in many cases, it depends on properties of the transmitting sensor node and of extraneous features such as terrain. For example, a sensor node may have a power budget that depends on a battery model, and the power it uses for transmission will affect the range.

The mysterious argument called *properties* plays a central role. This argument is used to specify (model-dependent) information about a particular transmission. The properties argument is always a RecordToken, which is a composite data type in Ptolemy II that has named fields of arbitrary type. The AtomicWirelessChannel base class provides a defaultProperties parameter that defines the fields that are relevant for a particular channel.

A simple use of the properties field would be to specify the transmission range for a particular transmission. Indeed, the LimitedRangeChannel subclass of WirelessChannel, has a defaultProperties value of {range = infinity}. A user of this channel could change this to, for example, {range = 100.0}, to represent that by default, transmissions have a range of 100 meters. An individual transmission can override this by setting the outsideTransmitProperties parameter of the sending port.

This model, however, is still simplistic. Communication ranges are typically not simple distances. More realistic models are supported by the PowerLossChannel subclass. This class has a parameter *powerLossFactor* whose default value is the expression "*efficiency* / (4 * PI * *distance* * *distance*)," which assumes that the transmit power is uniformly distributed on a sphere of radius *distance*. The variables *distance* and *efficiency* (which is intended to represent antenna efficiency) are convenience variables provided in the scope in which this expression is evaluated. The user of this model may replace this expression with any expression using the rich Ptolemy II expression language. The channel will then calculate the receive power using the specified *powerLossFactor* and provide the receive power to the receiving node via the `getProperties()` method of its input ports. The receiving node can then determine whether the signal has enough power to be received.

Much more sophisticated propagation models can be encapsulated and made available to the community as reusable components.

4.4 Antenna Gains and Terrain Models

Using the API as described so far, there appears to be no mechanism for implementing antenna gains or terrain models. These depend on the signal path from the transmitter to the receiver. However, close inspection reveals that the API is rich enough to accommodate these. In particular, the

WirelessChannel interface has a key method, `registerPropertyTransformer()`, which can be used to register any object that implements the TransformProperties interface (which includes any object that implements WirelessChannel). An object that implements this interface is given the opportunity to modify the transmit properties of any transmission (or it can selectively indicate an interest only in transmissions coming from a particular port).

A transmit antenna model, for example, can be realized by an object that implements the TransformProperties interface. In fact, we can use Ptolemy II infrastructure to provide an object that uses another Ptolemy II model to implement the property transformation. Thus, the full suite of sophisticated signal processing capabilities of Ptolemy II are at the disposal of the builder of the antenna model.

The same goes for terrain models, although there is a caveat. Property transformers are required to implement modifications of the properties record that are commutative. That is, if there are several property transformers that can affect a particular transmission, the result of applying these transformers needs to be the same regardless of the order in which they are applied. For simple terrain models that apply only power loss, this will often be true. For some more sophisticated terrain models, however, it will not be true. Such models must be implemented as channels, subclassing for example the PowerLossChannel.

4.5 Delay Channels

The DelayChannel subclass of ErasureChannel has a *propagationSpeed* parameter that the channel uses to determine the delay between transmission and reception of a signal. In the DelayChannel class, when the `transmit()` method is called, the channel calculates the delay to the specified location and requests that the director re-invoke it after that delay has elapsed (by calling the `fireAt()` method of the director, which places a request on the event queue). When it is reawakened, it delivers the message to the receiver.

4.6 Message Duration and Collisions

A message duration can be represented by a properties record with a *duration* field. We have provided a simple actor that can be put into a receiver node that uses this field to determine whether two messages have collided. It also uses a *power* field and checks the ratio of received powers against an *SNRThresholdInDB* parameter. This actor has two outputs, one of which produces a received token whose power to interference ratio exceeded the specified threshold, and the other of which produces any received tokens that did not meet this threshold. Thus, it is easy to build a model that monitors collisions, and to implement retransmission protocols in the event of collisions. The finite state machine domain of Ptolemy II is particularly convenient for this.

4.7 Battery Models and Modal Models

Sensors nodes have limited resources and cannot operate for an infinite amount of time. Predicting sensor node failures is important in creating a network model. Some factors that can cause a network failure are battery power, geographical changes that can cause sensor nodes to go out of range, signal strength, and processing problems. Models of the state of a node can be constructed using the facilities of Ptolemy II, including finite state machines to represent the state of the node and numerical models of battery life or energy scavenging.

5. FRAMEWORK INFRASTRUCTURE

The Ptolemy II framework provides some useful infrastructure.

5.1 Hierarchy and Heterogeneity

Ptolemy II supports hierarchical mixing of distinct models of computation. The example in Figure 2 hints at this capability, since the inside model and the outside model have distinct directors. In this case, both directors implement discrete-event semantics, with only small differences. However, it is possible to have much bigger differences. It is also not uncommon for the director inside to be an instance of the same class as the director outside. This permits a certain modularity in design.

In order to support this, the WirelessIOPort class in Figure 4 can optionally specify both an *insideChannel* and an *outsideChannel*. If the outside channel is specified, then wireless communication is used on the outside. If the inside channel is specified, then wireless communication is used on the inside. Both can be used at the same time.

Another useful combination uses the continuous-time domain of Ptolemy II. This domain includes a CTDirector with a sophisticated numerical solver for ordinary differential equations and extensive support for hybrid systems modeling [2]. This can be used, for example, to construct sophisticated models of the physical mobility of mobile sensor platforms.

5.2 Visualization

Ptolemy II 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. Examples that use this visualization capability are described in section 6.

5.3 Type System

Ptolemy II includes a sophisticated type system [20]. 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.

Note, however, that it would be unwise to explicitly model type constraints between every transmitter and every receiver using a channel. If there are n such users, this would be n^2 constraints, which for large n could bog down type resolution. As shown in Figure 4, a channel contains a single port, an instance of ChannelPort. This is used to set up n type constraints, one to each user of the channel. This simplifies type resolution and

keeps the static analysis of the model tractable even for large models.

6. APPLICATIONS

Although our experiments are still preliminary, we have constructed some non-trivial examples to illustrate the use of the infrastructure described here. We cannot describe them all here, but these are illustrative of the capabilities.

6.1 Flooding

First, [4] describes an evaluation of an algorithm for efficiently broadcasting queries in a sensor network. To reach all nodes, when a sensor receives a radio message, it may repeat the message. The objective is to reduce the number of such repetitions required to permeate the network. A screen image of one of the experiments performed is shown in Figure 5. In that image, each circle represents a sensor node. At the center of the circle is an icon for an antenna. This represents the location of the transmitter and receiver for the node. The sensor nodes are randomly scattered (an actor is provided in the library to realize the random scatter).

An execution of this model propagates a fixed number of queries, specified as a model parameter, through the sensor network. In this experiment, the base node, shown with a green translucent circle in Figure 5, broadcasts a query periodically. Initially all nodes repeat this query upon hearing it. However, as the execution proceeds, the nodes learn about the network topology around them, and based on the distributed algorithm described in [4], decide to continue repeating or to stop repeating. The nodes with red icons in Figure 5 are those that repeat, while the nodes with blue icons do not repeat, after the algorithm has converged.

In this experiment, the sensor nodes communicate via a radio channel with no time delay and no noise or losses. The broadcast range of a sensor node is a parameter of the node, and is indicated by the circular icon. In Figure 5, it is identical for all nodes; however, it can easily be varied. A more sophisticated channel model can be used by simply dropping it into this model and renaming it to match the channel name specified by the nodes.

6.2 Triangulation

The second experiment, described in [1], is shown in Figure 6. In this experiment, a model of the power source (batteries, in this case) of sensor nodes is coupled with a model of a scenario where acoustic sensors cooperate to determine the location of a sound source by triangulation.

As in the SensorSim simulator [17], multiple channels can be used in the same model in VisualSense. In this second experiment, an acoustic channel carries signals from a mobile sound source, and sensor nodes communicate via a radio channel. The acoustic channel includes propagation delay, while the

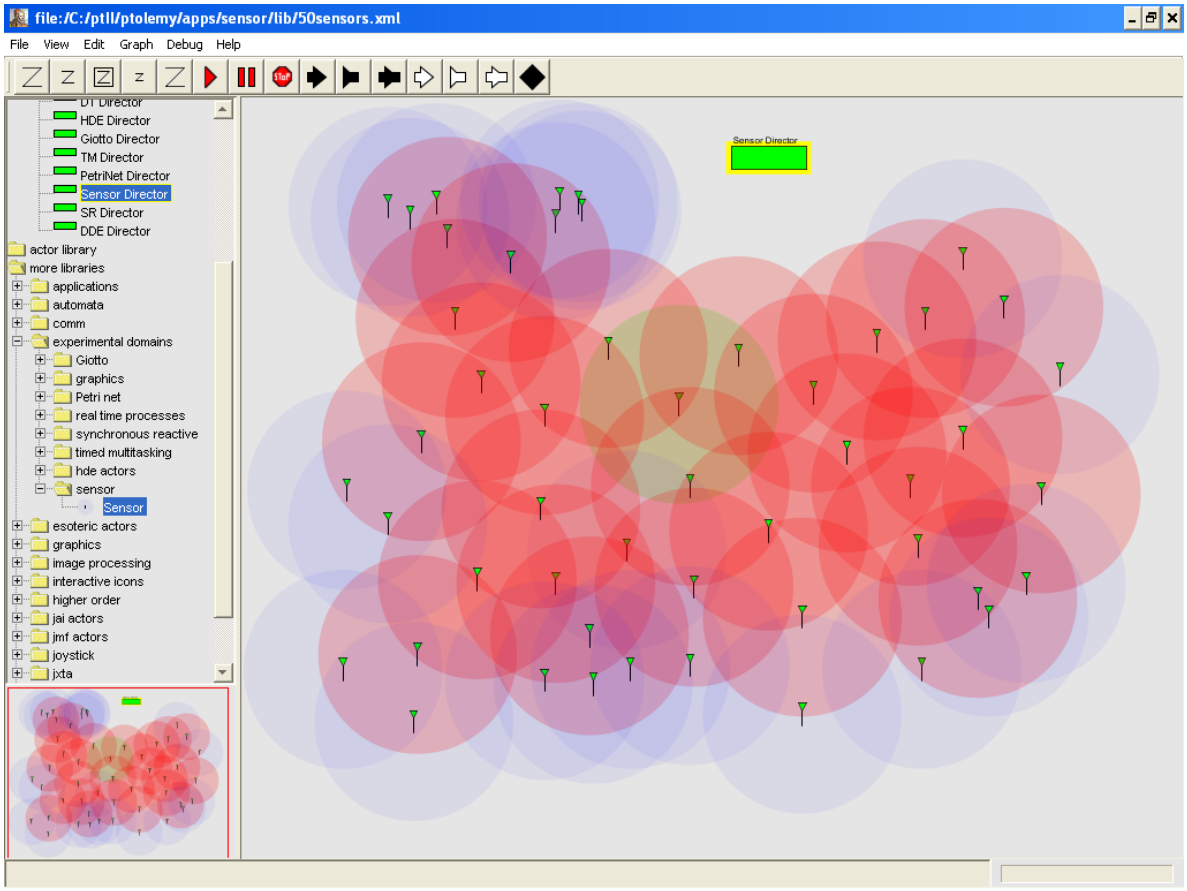

Figure 5. A screen image of a model described in [4] that is used to evaluate a flooding communication optimization heuristic.

radio channel is modeled as instantaneous, similar to the model in the flooding experiment described above. As with that experiment, the circular icons of the nodes represent their transmit range.

The acoustic channel carries signals from a mobile sound source to the sensor nodes. When a sensor node detects an acoustic signal, it broadcasts the observation (the time of the detection and the location of the sensor node) via the radio channel. A sound tracker actor collects the observations and computes the time and location of sound emissions from the mobile source.

6.3 Traffic Monitoring

Most sensor networks are highly dynamic. Objects monitored by a sensor network come and go, and may move around in the sensor field. New sensor nodes can join the network, and when the battery of a sensor node is depleted, it leaves the network.

A natural model of such dynamic sensor networks must support changes not only to the interconnection topology, but also to the set of components present in a network. As mentioned before, Ptolemy II has support for mutations of the model structure. We illustrate how this can be leveraged in sensor network modeling with a traffic monitoring application.

Sensors are distributed on a stretch of road to collect traffic information that is sent to a base station for further analysis. To model and simulate such a network, we first construct a model for the sensor field, which includes a component for each sensor node and the channels (a wireless channel for communication between sensors, and an acoustic channel for the signal propagation from the passing vehicles to the sensors).

In order to simulate the behavior of the network, we also need a stimulus model to generate traffic inputs to the sensor field. The question is what kind of inputs the stimulus model should provide to the sensor field model. Intuitively it is a car that enters the stretch of road on one boundary. In this sense, the stimulus is actually a car model added to the sensor field. Since cars can enter the area at any time and then leave after a while, it is not feasible to include them statically in the sensor field model.

To capture the dynamic structure due to cars entering or leaving, we use a higher order actor (one that takes as input another model). Such an actor contains another model that specifies its computation, and during execution, the contained model can be dynamically changed. The higher-order actor has two inputs, with the first input receiving data that the contained model should process, and the second input receiving model changes to the current contained model. The model changes can include

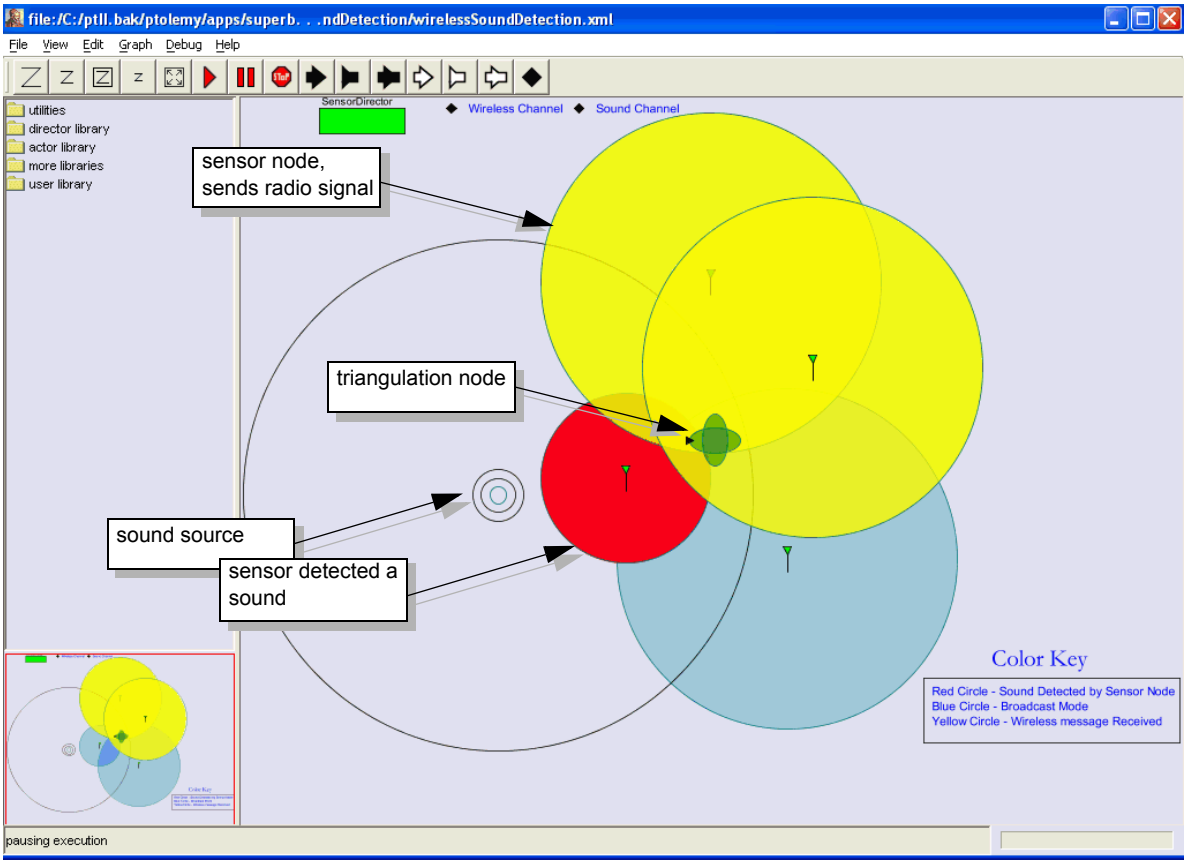

Figure 6. Example of a Ptolemy II model from [1] showing a sound source that moves as the model executes, four sensors that detect the sound and communicate via radio, and a sound tracker that receives the radio signals and uses triangulation to identify the location of the sound source.

adding new components such as actors that model cars, removing existing components, and adding or removing connections.

When execution begins, the higher-order actor contains an empty model inside. It first receives a model change for constructing the sensor field. After the change is applied, the execution continues with no vehicles in the sensor field. When the traffic model decides that there is a car entering the field, it generates a car model and sends it to the higher-order actor, which then changes the contained model to include the car model. Execution continues with the car moving in the area according to its driving program, and sensors on its path detect whether there is a passing car by listening to the sound channel. If it detects a car, the sensor then sends the data to the base station actor.

7. CONCLUSION

We have described an extensible software framework for sensor network modeling called VisualSense that is built on the Ptolemy II framework. Both VisualSense and Ptolemy II are open-source software, freely available at http://ptolemy.eecs.berkeley.edu. We hope that the community can use this framework to encapsulate and exchange methods and expertise in channel modeling and sensor node design.

8. ACKNOWLEDGMENTS

This research is part of the Ptolemy project, which is supported by the National Science Foundation (NSF award number CCR-00225610), and Chess (the Center for Hybrid and Embedded Software Systems), which receives support from the State of California MICRO program, and the following companies: Daimler-Chrysler, Hitachi, Honeywell, Toyota and Wind River Systems.

9. REFERENCES

[1] P. Baldwin, "Sensor Networks Modeling and Simulation in Ptolemy II," UC Berkeley, August 8, 2003.

[2] A. Cataldo, C. Hylands, E. A. Lee, J. Liu, X. Liu, S. Neuendorffer and H. Zheng, "Hyvisual: A Hybrid System Visual Modeler," Technical Memorandum UCB/ERL M03/30, University of California, Berkeley, July 17, 2003.

[3] A. Chakrabarti, L. de Alfaro and T. A. Henzinger, "Resource Interfaces," EMSOFT, Philadelphia, PA, LNCS 2855, Springer, October 13-15, 2003.

[4] C. T. Ee, N. V. Krishnan and S. Kohli, "Efficient Broadcasts in Sensor Networks," Unpublished Class Project Report, UC Berkeley, Berkeley, CA, May 12, 2003.

[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*, **91**(2), January, 2003.

[6] J. Elson, S. Bien, N. Busek, V. Bychkovskiy, A. Cerpa, D. Ganesan, L. Girod, B. Greenstein, T. Schoellhammer, T. Stathopoulos and D. Estrin, "Emstar: An Environment for Developing Wireless Embedded Systems Software," CENS Technical Report 0009, Center for Embedded Networked Sensing, UCLA, March 24, 2003.

[7] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1994.

[8] A. Girault, B. Lee and E. A. Lee, "Hierarchical Finite State Machines with Multiple Concurrency Models," *IEEE Transactions On Computer-aided Design Of Integrated Circuits And Systems*, 18(6), June 1999.

[9] C. Hylands, E. A. Lee, J. Liu, X. Liu, S. Neuendorffer, Y. Xiong and H. Zheng, "Heterogeneous Concurrent Modeling and Design in Java: Volume 1: Introduction to Ptolemy II," Technical Memorandum UCB/ERL M03/27, University of California, Berkeley, CA USA 94720, July 16, 2003.

[10] C. Hylands, E. A. Lee, J. Liu, X. Liu, S. Neuendorffer, Y. Xiong and H. Zheng, "Heterogeneous Concurrent Modeling and Design in Java: Volume 3: Ptolemy II Domains," Technical Memorandum UCB/ERL M03/29, University of California, Berkeley, CA USA 94720, July 16, 2003.

[11] T. J. Kwon and M. Geria, "Efficient Flooding with Passive Clustering (Pc) in Ad Hoc Networks," *ACM SIGCOMM Computer Communication Review*, **32**(1), January, 2002.

[12] E. A. Lee, "Modeling Concurrent Real-Time Processes Using Discrete Events," *Annals of Software Engineering* **7**: 25-45, March 4th 1998.

[13] M. Löbbers, D. Willkomm, A. Köpke, H. Karl, "Framework for Simulation of Mobility in OMNeT++ (Mobility Framework)," February 09 2004, http://www.tkn.tu-berlin.de/research/research_texte/framework.html.

[14] The CMU Monarch Project, "The CMU Monarch Project's Wireless and Mobility Extensions to NS," 1998 (see also http://www.monarch.cs.cmu.edu/)

[15] Ns-2, http://www.isi.edu/nsnam/ns, 2004.

[16] OPNET Technologies, Inc., "OPNET Modeler," http://opnet.com/products/modeler/home.html, 2004.

[17] S. Park, A. Savvides and M. B. Srivastava, "Sensorsim: A Simulation Framework for Sensor Networks," *3rd ACM International Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, Boston, Massachusetts, United States, ACM Press, August 20, 2000 (see also http://nesl.ee.ucla.edu/projects/sensorsim/).

[18] H.-Y. Tyan, "Design, Realization and Evaluation of a Component-Based Compositional Software Architecture for Network Simulation," Ph.D. Dissertation, Ohio State University, 2002. (see also http://www.j-sim.org)

[19] A. Varga, "The Omnet++ Discrete Event Simulation System," *Proceedings of the European Simulation Multiconference* (ESM'2001), Prague, Czech Republic, June 6-9, 2001 [see also http://www.omnetpp.org/].

[20] Y. Xiong, "An Extensible Type System for Component-Based Design," Technical Memorandum UCB/ERL M02/13, University of California, Berkeley, CA 94720, May 1, 2002.