Multimodeling: A Preliminary Case Study

Christopher Brooks
Thomas Huining Feng
Edward A. Lee
Reinhard von Hanxleden

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2008-7
http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-7.html

January 17, 2008

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 awards #CCR-0225610 (ITR),#0720882 (CSR-EHS: PRET), #0647591 (CSR-SGER), and #0720841 (CSR-CPS)), the U. S. Army Research Office (ARO #W911NF-07-2-0019), the U. S. Air Force Office of Scientific Research (MURI #FA9550-06-0312 and AF-TRUST #FA9550-06-1-0244), the Air Force Research Lab (AFRL), the State of California Micro Program, and the following companies: Agilent, Bosch, DGIST, Lockheed Martin, National Instruments, and Toyota.

Multimodeling: A Preliminary Case Study *

Christopher Brooks
UC Berkeley
cxh@eecs.berkeley.edu

Thomas Huining Feng
UC Berkeley
tfeng@eecs.berkeley.edu

Edward A. Lee
UC Berkeley
eal@eecs.berkeley.edu

Reinhard von Hanxleden
University of Kiel
rvh@informatik.uni-kiel.de

January 17, 2008

Abstract

We take a pre-existing Statecharts model of a simple traffic light controller and re-implement it in Ptolemy II. This exercise reveals that Statecharts can be usefully conceptualized as a hierarchical combination of two distinct models of computation (MoCs), finite state machines (FSMs) and synchronous/reactive (SR). Once conceptualized this way, we can add additional MoCs to the mix. We illustrate this by adding a discrete-event (DE) model of the environment in which the traffic light operates. We then construct a second model of a deployment of the system on two microcontrollers communicating wirelessly, showing that we can effectively leverage both DE and an extension in Ptolemy II that supports modeling of wireless communication networks. This exercise reveals that even though the original model was intended to be a purely functional model, it in fact imposes constraints on the implementation. The model had to be refactored to get a distributed deployment model. Finally, we show that the portions of the models defining the control logic of the lights can be shared between the functional and deployment models using actor-oriented classes. This eases maintenance of the models.

1 Introduction

Multimodeling is the act of combining diverse models. We consider two forms that this can take. In the first form, hierarchical compositions of distinct modeling styles are combined to take advantage of the unique capabilities and expressiveness of the distinct modeling styles. In the second form, distinct and separate models of the same system are constructed to model different aspects of the system. We call the first form **hierarchical multimodeling** and the second form **multi-view modeling**.

This paper starts with a simple traffic light controller given as a Statecharts model, and interprets it as a hierarchical multimodel. We then show an equivalent model constructed with Ptolemy II [10] that is much more explicit about the fact that this is a hierarchical multimodel that combines two distinct modeling formalisms (state machines and

*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 awards #CCR-0225610 (ITR),#0720882 (CSR-EHS: PRET), #0647591 (CSR-SGER), and #0720841 (CSR-CPS)), the U. S. Army Research Office (ARO #W911NF-07-2-0019), the U. S. Air Force Office of Scientific Research (MURI #FA9550-06-0312 and AF-TRUST #FA9550-06-1-0244), the Air Force Research Lab (AFRL), the State of California Micro Program, and the following companies: Agilent, Bosch, DGIST, Lockheed Martin, National Instruments, and Toyota.

synchronous/reactive models). We next show that the model can be considerably enriched with a third modeling formalism, discrete events, something that is not possible using Statecharts alone.

The resulting model is a complete model of the functionality of a simple traffic light and the environment in which it operates. But it is only a model of the functionality.

We continue by constructing a **deployment model**, which models a hardware implementation (in our example, two components of the traffic light communicate wirelessly). The deployment model and the functional model together are a form of multi-view modeling. The distinct models share certain components. We show that we can use actor-oriented classes [22] to maintain consistency across these distinct models. That is, as the models evolve, their shared components remain identical.

We show that the process of constructing the deployment model reveals that the original functional model is not, in fact, agnostic about implementation. Particular choices made in the functional model are in fact inconsistent with the wireless deployment that we selected. Maintaining orthogonality between models in multi-view modeling is challenging. As additional models are constructed for distinct views, refactoring of previously constructed models invariably becomes necessary.

2 Related Work

One of the key innovations in Statecharts [16] is the introduction of concurrency ("and states"), as well as hierarchy, to state machine models. There exist several variants of Statecharts, which differ in the precise timing and concurrency models and other aspects such as possible reaction to signal absence; von der Beeck [4] compares 21 dialects, and since then numerous other variants have been developed, such as Simulink/Stateflow and UML Statecharts. We here generally presume a "fully synchronous" semantics of Statecharts, as embodied in SyncCharts [1], also called Safe State Machines (SSMs) [2], and use the SyncChart graphical syntax. However, for the case study discussed here, other semantics such as Harel's original semantics or the Stateflow semantics produce equivalent results.

Statecharts can be viewed as a hierarchical combination of a synchronous/reactive (SR) concurrent model of computation (MoC) [5] and a state machine model. This is a form of hierarchical multimodeling. This idea has been generalized, showing that other concurrent MoCs can be usefully combined with state machines [12]. That work followed on Ptolemy Classic [8], which provided a software architecture supporting a general form of hierarchical multimodeling. In [8], Buck et al. showed how to apply hierarchical multimodeling in applications that combined networking and signal processing. Hierarchical multimodeling has also been elaborated in ForSyDe [18], SPEX [24], and ModHelX [15]. A non-hierarchical approach to multimodeling is provided by Metropolis [13] and Colif [9]. This approach does not segregate distinct models of computation hierarchically.

Ptolemy Classic [8] also illustrated multi-view modeling applied to hardware/software codesign. One model specifies functionality and one specifies hardware architecture. This concept has been elaborated into a sophisticated methodology for hardware/software codesign called Y Charts [20], and has been developed into design tools like Metropolis [13]. Multi-view modeling has also formed a centerpiece of model-integrated computing [27] and has been applied in a number of large-scale system designs [14].

There is a third form of multimodeling, where a single model can be specialized to multiple distinct implementations [25]. That form is not discussed in this paper.

In this paper, we use a simple traffic light controller to illustrate the issues in multi-view modeling. A similar approach is taken by Feng, Zia, and Vangheluwe [11]. Our approach was also influenced by Huang [17].

In this paper, we leverage actor-oriented classes [22, 19, 21] to maintain consistency across multi-view models.

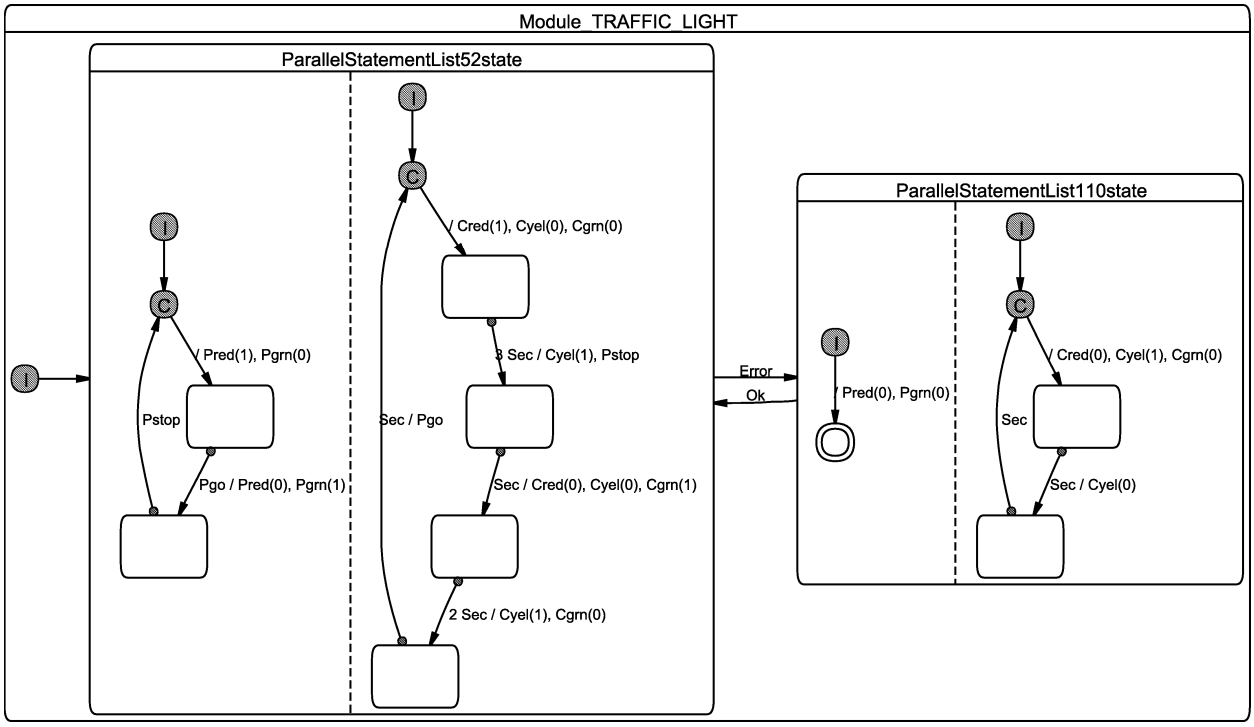

Figure 1: A Statecharts model of a simple traffic light controller.

3 The Traffic Control Model

A Statecharts model [16] of a simple traffic light controller is shown in figure 1. The module TRAFFIC_LIGHT has two states. The left state (which is also the initial state, as indicated by the "I" bubble pointing into it) represents normal operation, and the right state represents error condition operation. The transition to the error state is triggered by an unspecified external event called *Error*, and the transition back to the normal state by another external event called *Ok*.

Each of the normal and error states contains two concurrent state machines, one governing the operation of a pedestrian light and the other governing the operation of a car light. To determine the initial state of the pedestrian light, follow from the "I" bubble through the "C" (connector) bubble along the arc labeled `/Pred(1), Pgrn(0)`. The (unlabeled) state at the end of that arc is the initial state.

In Statecharts, arcs are labeled with `guard/action`, where the guard specifies the conditions under which the transition is taken. The label `/Pred(1), Pgrn(0)` does not include a guard, so the transition is taken unconditionally. The action `Pred(1), Pgrn(0)` assigns to variable `Pred` (for "pedestrian red light") the value 1, which we interpret to mean to illuminate the pedestrian red light. The pedestrian green light is turned off by the action `Pgrn(0)`.

The transition labeled `Pgo / Pred(0), Pgrn(1)` is triggered by the event `Pgo` (for "pedestrian go") and turns off the red light and on the green.

The transition labeled `Pstop` specifies only a guard, no action. This transition is triggered by the signal `Pstop`,

and then proceeds instantaneously through the connector bubble and along the transition labeled `/Pred(1), Pgrn(0)`, which specifies the action to take.

The concurrent car control state machine is to the right of the dashed line that separates it from the pedestrian light control state machine. This machine has four states, and using a similar notation, turns on and off red, yellow, and green lights for the cars. For example, the action `Cred(1)` turns on the car red light.

The car state machine has an additional notation where a guard like `2 Sec` is given. This specifies that the transition is triggered after remaining in the previous state for two consecutive instances of the event `Sec`. This event is supplied by the environment, in this case to indicate the passage of one second. Note that at this level we do not explicitly model physical time as part of our semantics, but instead assume it is supplied by the environment. This differs from Harel's original Statechart dialect, which did include a timeout mechanism as part of the language. Instead, we adopt the *multiform notion of time*, where the passage of time is seen just as any other event, such as passage of distance, and is handled with the same mechanisms [7].

The error state on the right can now be easily read, knowing the notation. It turns off both the red and the green pedestrian lights, and blinks the car yellow light.

4 Hierarchical Multimodeling

Statecharts can be viewed as a hierarchical combination of synchronous/reactive (SR) models and finite state machines (FSMs). The "and states" compose components according to the SR concurrency model, while the "or states" describe classical FSMs.

In Ptolemy II, this hierarchical combination of MoCs is more explicit. A Ptolemy II model equivalent to that of figure 1 is shown in figure 2. The top level of the hierarchy, labeled "TrafficLight," just like figure 1, shows two states, "Normal" and "Error." Unlike the Statecharts model, this figure also shows explicitly the three external inputs that need to be provided (Sec, Error, and Ok). In the graphical part of the Statecharts model, one can only infer from the fact that these events are mentioned and not asserted anywhere that they are in fact external inputs. The Ptolemy II model also explicitly shows five outputs (Pred, Pgrn, Cred, Cyel, and Cgrn), which control whether the pedestrian and car lights are on or off.

The transitions between the normal and error states are labeled with guards, `Ok_isPresent` and `Error_isPresent`. This means that these transitions are triggered by the presence of these externally provided events.

The normal and error states have refinements, shown in the boxes labeled "Normal" and "Error." These refinements inherit the input and output ports from the TrafficLight model. The refinements, however, have a different model of computation. Their MoC is indicated by the **director**, labeled "SR Director," which specifies a synchronous/reactive MoC.

In Ptolemy II, states in state machines can have refinements, as in TrafficLight, and hence are called **modal models** [12] to distinguish them from classical state machines. This corresponds to the hierarchy in Statecharts. The states represent modes of operation, and in each mode, the behavior is given by the refinement. In Ptolemy II, the semantics are as follows. When a modal model fires (which in the SR MoC occurs on each "tick" of a global clock [5]), the refinement of the current state is executed, then the guards on all outgoing transitions are evaluated. If exactly one of those guards is true, then the transition is taken and the actions on the transition are executed. If more than one of the guards is true, then unless the transitions are marked to tolerate nondeterminacy, then an exception will be thrown. If the transitions are so marked, then one of the enabled transitions is chosen arbitrarily.

The Normal refinement itself contains two components. Since these will execute under the SR MoC, they are concurrent, corresponding to the "and states" of figure 1. Unlike figure 1, the communication between these concurrent components is explicit in the Ptolemy II model. In the Statecharts model, two concurrent state machines communicate if one issues an event that is used by the other (e.g. to trigger a transition). The fact that communication is occurring

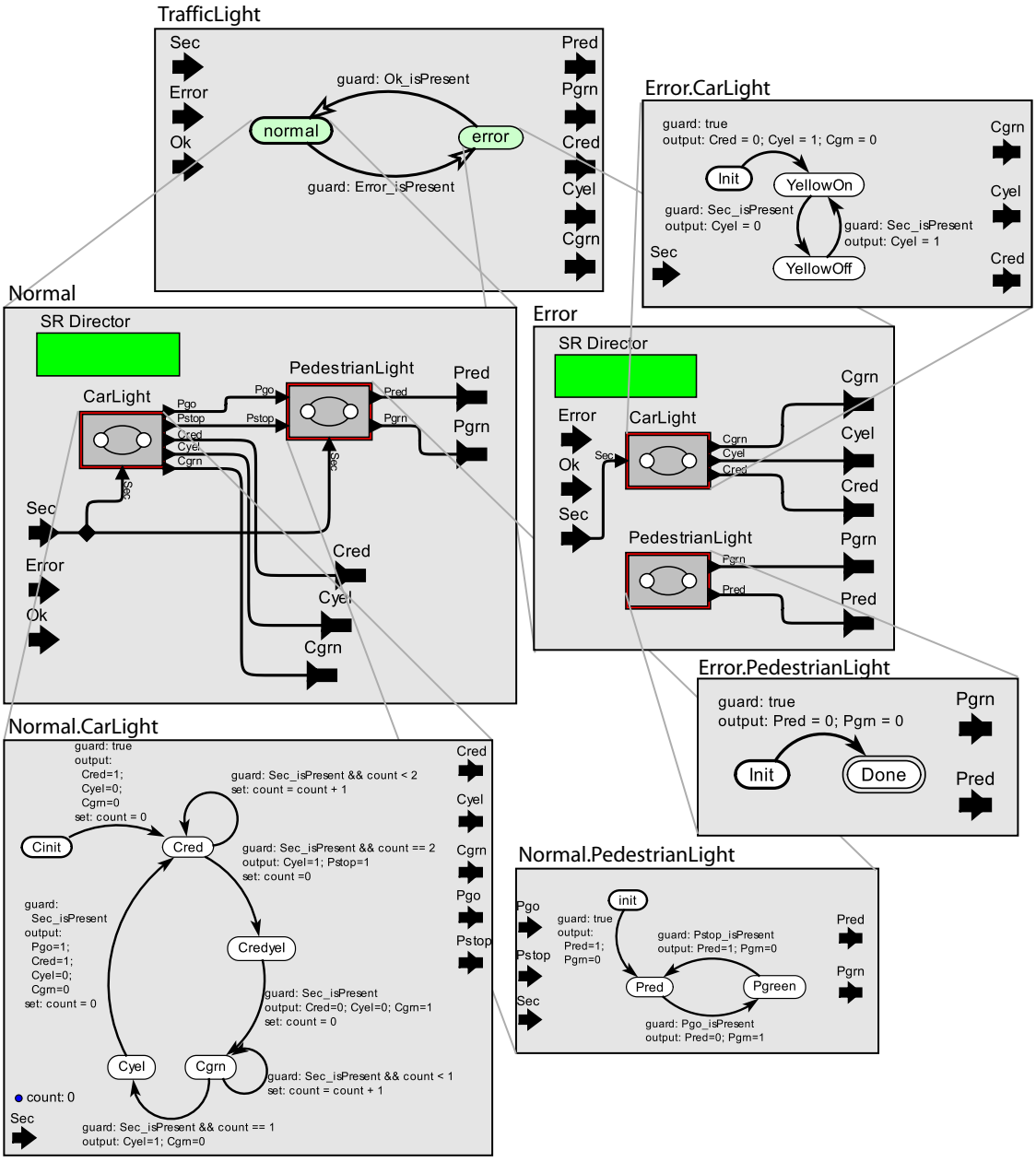

Figure 2: The traffic light controller in Ptolemy II.

must be inferred by the reader by matching the names of the events. In Ptolemy II, the names need not match; the communication is indicated instead by "wires" connecting the concurrent components. (In figure 2 the names on the wires connecting CarLight to PedestrianLight in Normal match only to correspond with the Statecharts model.)

There is an advantage to explicitly showing the connections between concurrent components, rather than relying on name matching. In particular, from looking at figure 1, it is very hard to tell whether there is feedback between the "and states." That is, it is hard to tell whether one and state issues an event e_1 in response to event e_2, where e_2 is issued in response to e_1. The fact that there is no such feedback is visible in the syntax in figure 2, but not in figure 1. Note that both SCADE and Esterel-Studio also support such an explicit syntax[6].

In 2, we see that Normal.CarLight provides the Pgo and Pstop events to Normal.PedestrianLight, and that otherwise, there is no communication between these concurrent components.

Normal.CarLight and Normal.PedestrianLight are themselves modal models, but where the states have empty refinements. These, therefore, are classical (extended) finite state machines. The extension is that in addition to a finite set of ordinary states (indicated by ellipses), the state machine can have local variables that have a value. In particular, Normal.CarLight has a variable `count`, shown at the lower left, which is used to measure the passage of time. Thus, unlike figure 1, there is no special syntax for triggers that consider multiple consecutive occurrences of an event. Instead, the variable `count` is updated in the actions on the transitions and evaluated in the guards.

Examining Normal.CarLight, we see that transitions all have a guard, which is an expression that when true triggers the transition. If the guard is "`true`," as it is on the transition between Cinit and Cred, then the transition will be taken on the first tick of the SR clock. Transitions with guard "`Sec_isPresent`" are triggered when the Sec input is present. This input is provided by the environment. Our design assumes that it is provided once per second.

The actions associated with each transition are divided into two categories, **output actions** and **set actions**. In the model of figure 2, there is no material difference between these because the SR model has no feedback. When an SR model has feedback, however, then at each tick of the SR clock there is an iteration to a fixed point. Such an iteration requires that components be able to assert output values without committing to state transitions. Since our example has no feedback, we do not discuss this further. You can assume that when a transition is triggered, both the output and set actions are executed.

In Normal.CarLight, you can see that output actions are used to turn lights on and off, while set actions are used to control the value of the count variable. You can also see that all guards are carefully defined to ensure that there is no nondeterminacy in the model. It is a convenient feature of Ptolemy II that modal models are checked (at run time) for nondeterminate behavior.[1]

At the top level (TrafficLight), when a transition is taken from normal to error or vice versa, the refinement of the destination mode is re-initialized. Thus, the state machines in Normal.CarLight, Normal.PedestrianLight, Error.CarLight, and Error.PedestrianLight will always start in their initial state (shown with a bold outline) when mode changes between normal and error occur. In the Ptolemy II syntax, this fact is indicated by the hollow arrowheads on the transitions in TrafficLight.

The only additional syntax needed to read figure 2 is the **final state** in Error.PedestrianLight, which is shown with a double ellipse. This is similar to the double circle in figure 1. In Ptolemy II, when this final state (labeled `Done`) is entered, the enclosing modal model (labeled `PedestrianLight` in the `Error` model) will henceforth be omitted from executions by the enclosing SR Director. That is, on subsequent ticks of the SR clock, the `PedestrianLight` actor will not be fired and its outputs will be deemed to be absent.

We can gain some insight by comparing the syntaxes of figures 1 and 2. In particular, we see that figure 2 is larger (more verbose). However, it is also more explicit about key information, such as whether there is communication between concurrent components. Moreover, it is much more literally hierarchical, and thus, at each level of the

[1] It would be better if this could be checked statically, but since guards and transitions involve arbitrary operations on variables, the question of whether a modal model is determinate is undecidable in general.

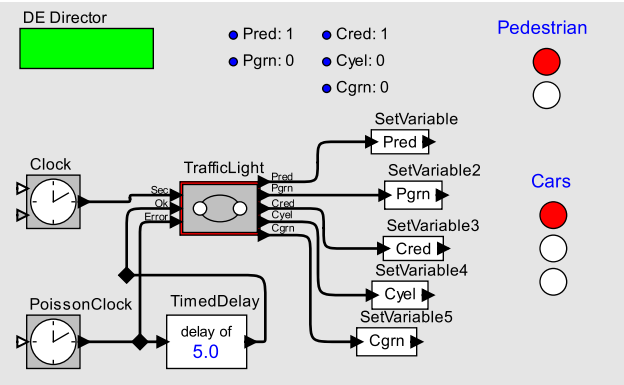

Figure 3: A DE model of the traffic light environment in Ptolemy II.

hierarchy, the diagram presents a simple abstracted view of the system. Such an explicitly hierarchical syntax may scale more easily to large models. Note that although the model in figure 1 uses a single sheet, Statechart tools typically allow to spread different hierarchy levels across several sheets as well, i.e., both of the parallel macro states could have been placed into separate sheets.

We consider these as different, equivalent views of a model. Ideally the modeler could freely choose among these. It is quite conceivable to have the modeling tool synthesize such views automatically, by transforming them into each other or by synthesizing them from a separate source. In fact, the Statechart in figure 1 has been synthesized from an equivalent, textual Esterel description [26], and it seems feasible to alternatively synthesize the model in figure 2.

5 Modeling the Environment

The top levels in both figures 1 and 2 present incomplete models. In particular, they both rely on external inputs to have any behavior at all. In the Statecharts model, if we were to provide a model of the environment that provides these signals, it would have to conform with the Statecharts semantics. Thus, we would be restricted to SR for concurrency and FSMs for state machines. Neither of these is really rich enough to correctly the model the environment for this problem.

In Ptolemy II, we are not constrained to SR for the concurrency model. In fact, many concurrent MoCs have been implemented in Ptolemy II, and many can be combined hierarchically with SR. For our purposes here, the most natural MoC for modeling the environment is discrete events (DE), implemented by the DE Director in Ptolemy II.

In the DE MoC, actors communicate by sending each other time-stamped events. The DE Director fires actors in time-stamp order. The semantics of DE is shown in [23] to be a generalization of the SR semantics. Because it is a generalization, the hierarchical combination of the two MoCs is clean and rigorously defined. Very simply, if a DE actor is fired in response to time stamped events, and that DE actor internally contains an SR model, then the SR model executes one “tick” of its clock.

A simple model of the environment for our traffic light example is shown in figure 3. The TrafficLight actor is the modal model shown in figure 2. It has three inputs, Sec, Error, and Ok. It will fire when any of these three inputs contains the oldest (least time stamp) event in the system. The Sec input is driven by a Clock actor, which produces an event once per second. The Error input is driven by a PoissonClock actor, which produces events according to a Poisson process. A parameter of that actor specifies the mean interarrival time of those events.

In our simple environment model, the Ok input receives an event five seconds after the error event occurs. This models a fixed (and unrealistic) repair time. But it is easy to see how more interesting models could be constructed.

The outputs of the TrafficLight actor are wired to instances of the SetVariable actor. These instances simply record the values in variables shown at the top of the diagram, Pred, Pgrn, Cred, Cyel, and Cgrn. The reason for doing this is that we can exploit a feature of Ptolemy II and create an interactive animation of the execution of this model. In particular, the circles on the right of the diagram have colors that set by expressions that depend on these variables. For example, the top-most circle has its fill color defined by the expression

```
(Pred == 1) ? {1.0, 0.0, 0.0, 1.0} : {1.0, 1.0, 1.0, 1.0}
```

This means that if the variable Pred has value 1, the color is red, and otherwise the color is white.[2] Thus, when executing the model, the circles change colors just as the lights in a traffic light would.

6 The Deployment Model

The model of figure 2 describes the functionality of the traffic light without particular concern for how it is implemented. Suppose that to implement this, two microcontrollers are used, one for the car light and one for the pedestrian light. Suppose further that these two microcontrollers communicate with one another via a wireless radio link. A model that describes this architecture is called a **deployment model**.

Such a model in Ptolemy II is shown in figures 4 and 5. The top level of this model uses the WirelessDirector, a generalization of the DE Director that supports the modeling of wireless communication systems [3]. Semantically, the Wireless MoC is identical to DE. Syntactically, communication between actors is indicated not by wires but by identifying a wireless channel model (labeled RadioChannel in the figure) that mediates the communication. In principle, the RadioChannel actor can model arbitrary properties of the radio link, including reliability, interference, and security properties. In the model we built, it models only the limited range of the radio communication.

At the top level, the CarLight and PedestrianLight actors have custom icons that contain circles that change color in response to changes in the values of their parameters. The single output port of the CarLight actor communicates via the wireless channel represented by the RadioChannel actor to the single input port of the PedestrianLight actor.

Inside the CarLight actor we see a model of the hardware of the car light. It includes a Clock actor that provides one clock tick per second, and a PoissonClock actor that models hardware faults. We observe that in this model, hardware faults need to be modeled at the local level, inside the CarLight model, whereas in figure 2 they are modeled at the systemwide level. Although this seemed reasonable when we constructed the functional model, because of the choice of deployment, where the car light and the pedestrian light are on distinct hardware platforms, that original choice proves awkward. There is no no global “normal” and “error” state; each component has to separately decide whether it is in the “normal” or “error” state. Maintaining coherence of these decisions turns out to be a fairly important aspect of the design, one that did not appear in the functional model. In fact, in our model, if the car light fails, it sends a failure message to the pedestrian light, which then turns off. However, if the pedestrian light fails, the car light continues operating normally. This is probably not a desirable property of the system.

A designer faces two choices: she could refactor the functional model in figure 2 to put the “normal” and “error” states lower in the hierarchy, or she could interpret the original design as an abstraction and create a distinct model for the deployment. Our choice was to do the latter. As a consequence, the two models share less of the design than they could.

Our decision, however, was to share as much as possible without modifying the original design. This reflects the practicalities of system design on a large team, where changing designs is not always possible.

[2] Colors are defined by an array R, G, B, A, specifying red, green, blue, and alpha values. When these values are 1.0, the corresponding color is fully present. When the alpha value is 1.0, the color is opaque.

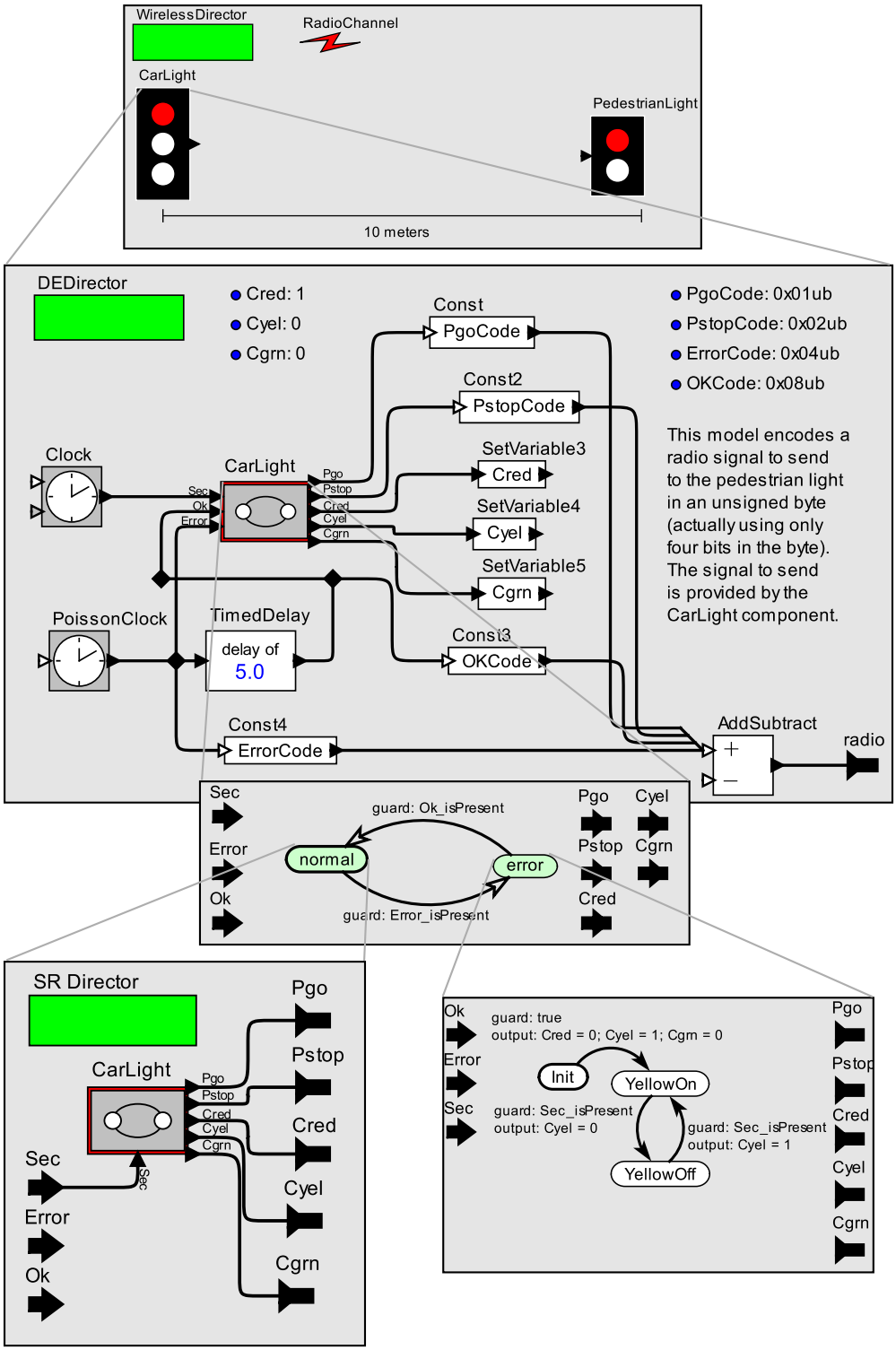

Figure 4: Wireless deployment, showing the car light model.

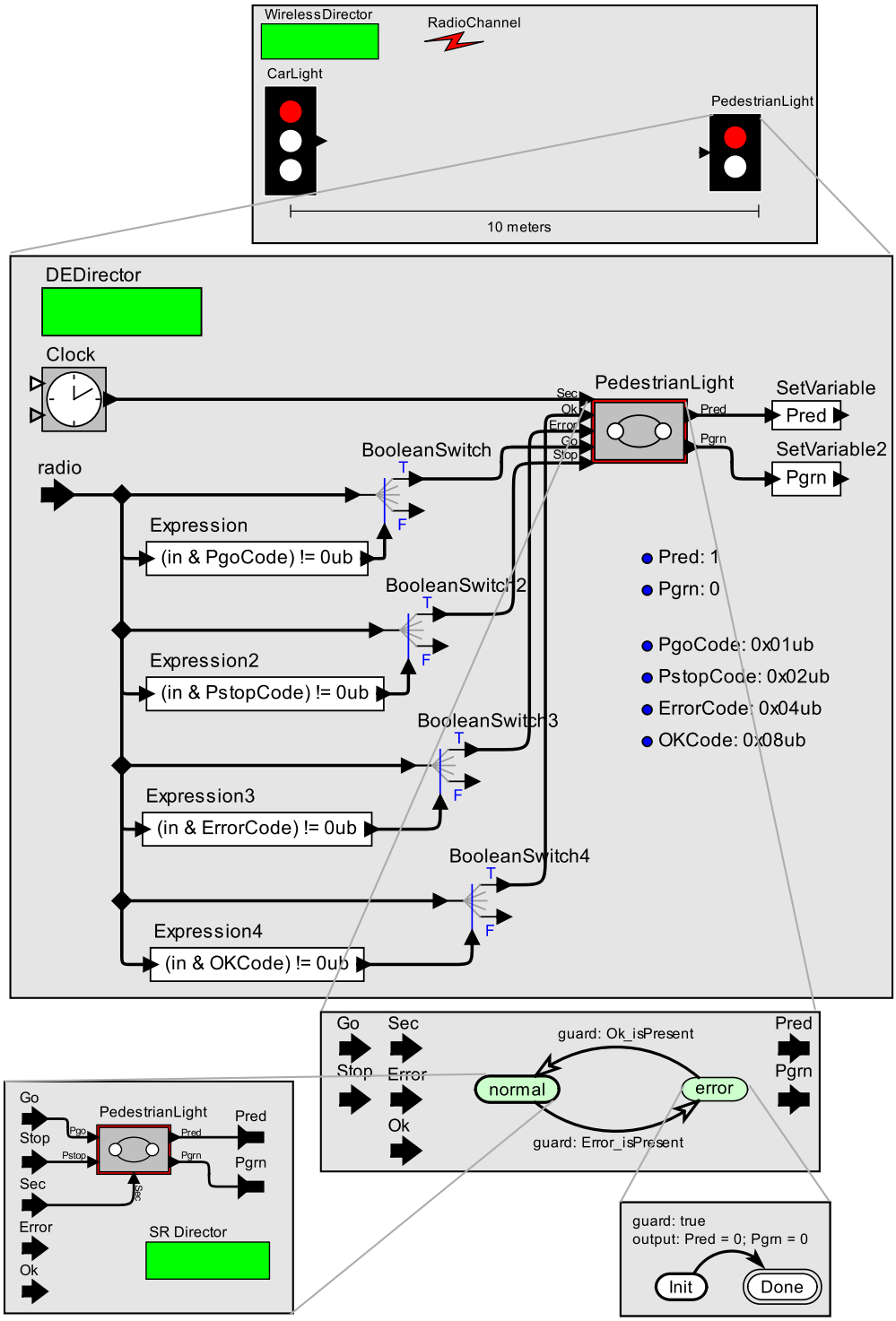

Figure 5: Wireless deployment, showing the pedestrian light model.

The original Statecharts model that we worked with was a European design, and consequently turned on red and yellow lights simultaneously before switching to green. What if we wanted to change the design to the American system, where we move directly from red to green? Is there any way this change could be made in one place and apply to both the functional model and the deployment model?

In fact, our model uses instances of two actor-oriented classes as implemented in Ptolemy II [22] to accomplish this objective. The CarLight and PedestrianLight actors in figures 4 and 5 are instances of the same model definitions labeled Normal.CarLight and Normal.PedestrianLight in figure 2. If any change is made to the internals of these actors, the change is reflected in all instances. Thus, to change from the European to the American system, the change only needs to be made in one place. It will apply to both the functional and the deployment model.

7 Conclusion

We have shown that the Statecharts model of computation can alternatively be conceptualized as a hierarchical combination of finite state machines and a synchronous/reactive concurrency model, instead of as a single monolithic MoC. Furthermore, we have demonstrated the benefit of augmenting the Statecharts model—in its original monolithic form or as a hierarchical combination of MoCs—with other MoCs, such as discrete events, enabling modeling of other parts of the system. We illustrate this by showing converting a Statecharts model of a traffic light controller into a Ptolemy II model, and then augmenting the model with a model of its environment.

We have further shown the construction of a "deployment model," which gives details of an implementation that uses wireless communications between two hardware processors. This model can be used, for example, to refine the design for robustness and safety in the face of impairments on the wireless channel. Our deployment model, however, reveals that the original functional model implicitly imposed constraints on the system that make this particular deployment difficult to achieve. In particular, its top-level state machine splits the entire system into a normal mode and an error mode. But with two distinct processors communicating wirelessly, achieving this global mode transition reliably would be very difficult. We had to refactor the model to put this split lower in the hierarchy.

Despite the refactoring, our deployment model was able to share critical pieces of functionality with the original model. In particular, the logic governing the mode transitions of the two lights (car and pedestrian) are defined in an actor-oriented class, and both the functional model and the deployment model use instances of those classes. Thus, if we change this logic in the class definition (e.g. to change from the European style of lights to the American), then the change automatically appears in both models.

8 Acknowledgements

We thank Claus Traulsen and Hauke Fuhrmann for very helpful suggestions.

References

[1] C. André. Syncharts: A visual representation of reactive behaviors. Technical Report RR 95–52, rev. RR (96–56), I3S, April 1996.

[2] C. André. Semantics of S.S.M (Safe State Machine). Technical report, Esterel Technologies, April 2003.

[3] P. Baldwin, S. Kohli, E. A. Lee, X. Liu, and Y. Zhao. Modeling of sensor nets in Ptolemy II. In *Information Processing in Sensor Networks (IPSN)*, Berkeley, CA, USA, April 26-27 2004.

[4] M. v. d. Beeck. A comparison of statecharts variants. In H. Langmaack, W. P. de Roever, and J. Vytopil, editors, *Third International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 128–148, Luebeck, Germany, September 19-23 1994. Springer-Verlag.

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

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

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

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

[9] W. O. Cesario, G. Nicolescu, L. Guathier, D. Lyonnard, and A. A. Jerraya. Colif: A design representation for application-specific multiprocessor socs. *IEEE Design & Test of Computers*, 2001.

[10] 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):127–144, 2003.

[11] T. H. Feng, M. Zia, and H. Vangheluwe. Multi-formalism modelling and model transformation for the design of reactive systems. In *Summer Computer Simulation Conference (SCSC)*, San Diego, CA, USA, 2007.

[12] 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):742–760, 1999.

[13] G. Goessler and A. Sangiovanni-Vincentelli. Compositional modeling in metropolis. In *Second International Workshop on Embedded Software (EMSOFT)*, Grenoble, France, October 7-9, 2002 2002. Springer-Verlag.

[14] Z. Gu, S. Wang, S. Kodase, and K. G. Shin. An end-to-end tool chain for multi-view modeling and analysis of avionics mission computing software. In *Real-Time Systems Symposium (RTSS)*, pages 78 – 81, December 2003.

[15] C. Hardebolle and F. Boulanger. Modhel'x: A component-oriented approach to multi- formalism modeling, October 2 2007.

[16] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[17] Y.-S. Huang. Design of traffic light control systems using statecharts. *The Computer Journal*, 49(6):634–649, 2006.

[18] A. Jantsch and I. Sander. Models of computation and languages for embedded system design. *IEE Proceedings on Computers and Digital Techniques*, 152(2):114–129, 2005.

[19] G. Karsai, M. Maroti, k. Ldeczi, J. Gray, and J. Sztipanovits. Type hierarchies and composition in modeling and meta-modeling languages. *IEEE Transactions on Control System Technology*, to appear, 2003.

[20] B. Kienhuis, E. Deprettere, P. v. d. Wolf, and K. Vissers. A methodology to design programmable embedded systems. In E. Deprettere, J. Teich, and S. Vassiliadis, editors, *Systems, Architectures, Modeling, and Simulation (SAMOS)*, volume LNCS 2268. Springer-Verlag, November 2001.

[21] P. Kinnucan and P. J. Mosterman. A graphical variant approach to object-oriented modeling of dynamic systems. In *Summer Computer Simulation Conference (SCSC)*, pages 513–521, San Diego, CA, July 15-18 2007.

[22] E. A. Lee, X. Liu, and S. Neuendorffer. Classes and inheritance in actor-oriented design. *ACM Transactions on Embedded Computing Systems (TECS)*, to appear, 2008.

[23] E. A. Lee and H. Zheng. Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems, October 2007.

[24] Y. Lin, R. Mullenix, M. Woh, S. Mahlke, T. Mudge, A. Reid, and K. Flautner. SPEX: A programming language for software defined radio. In *Software Defined Radio Technical Conference and Product Exposition*, Orlando, November 13-17 2006.

[25] R. v. Ommering, F. v. d. Linden, J. Kramer, and J. Magee. The Koala component model for consumer electronics software. *Computer*, pages 78–85, 2000.

[26] S. Prochnow, C. Traulsen, and R. v. Hanxleden. Synthesizing safe state machines from esterel.

[27] J. Sztipanovits, G. Karsai, and H. Franke. Model-integrated program synthesis environment. In *Symposium and Workshop on Engineering of Computer Based Systems (ECBS)*. IEEE, March 1996.