Model Engineering using Multimodeling [⋆]

Christopher Brooks[1], Chih-Hong Cheng[2], Thomas Huining Feng[1], Edward A. Lee[1], and Reinhard von Hanxleden[3]

[1] University of California at Berkeley (cxh,tfeng,eal@eecs.berkeley.edu)
[2] National Taiwan University (patrickjeng@gmail.com)
[3] University of Kiel (rvh@informatik.uni-kiel.de)

Abstract. We study the simultaneous use of multiple modeling techniques in the design of embedded systems. We begin with a pre-existing Statecharts model of a simple case study, a traffic light for a pedestrian crossing, using it to illustrate the need for multimodeling and the pitfalls. The original model combines two distinct models of computation (MoCs), finite state machines (FSMs) and synchronous/reactive (SR). We add an additional MoC, a discrete-event (DE) model of the environment in which the traffic light operates, including a simple fault model, yielding a model that combines three different modeling techniques. We construct a second model of a hardware deployment and a third model that is an abstraction used for formal verification. The result is that this simple example uses three distinct models of the system (functional, deployment, verification), two of which hierarchically combine distinct modeling techniques (DE, SR, FSM). This exercise reveals some pitfalls of model-based design where multiple models are needed as well as some of the opportunities.

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. This model is simple

[⋆] 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 #0720882 (CSR-EHS: PRET) 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), the Air Force Research Lab (AFRL), the State of California Micro Program, and the following companies: Agilent, Bosch, HSBC, Lockheed-Martin, National Instruments, and Toyota.

enough to be fully described in a short paper, and yet rich enough to illustrate many major design issues. We first show an equivalent model constructed with Ptolemy II [13] that is much more explicit about the fact that this is a hierarchical multimodel that combines two distinct modeling formalisms (state machines and 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 hierarchical multimodel of the functionality of a simple traffic light interacting with the environment in which it operates.

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 [26] 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 agnostic about implementation. Particular choices made in the functional model are 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.

A third kind of model is often required for safety-critical embedded systems. In particular, the functional and deployment models may be too detailed for effective use of formal verification techniques such as model checking, and may include complicated temporal dynamics that are difficult to handle. We illustrate an *applied verification* technique where we synthesize from the functional model using code generation technology an SMV (Symbolic Model Verifier) model that can be used to check safety properties via model checking. This reveals that while the functional model satisfies a key safety requirement, the refactored deployment model does not.

2 Related Work

One of the key innovations in Statecharts [19] 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 [5] compares 21 dialects, and since then numerous other variants have been developed, such as Simulink/Stateflow and UML Statecharts. We here presume a "fully synchronous" semantics of Statecharts, as embodied in SyncCharts [2], also called Safe State Machines (SSMs) [3], 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 mix of a state machine model and a synchronous/reactive (SR) concurrent model of computation (MoC) [6]. This is a form of hierarchical multimodeling. This idea has been generalized, showing that other concurrent MoCs can be usefully combined with state machines [15]. That work followed on Ptolemy Classic [9], which provided a software architecture supporting a general form of hierarchical multimodeling. In [9], 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 [22], SPEX [28], and ModHelX [18]. A non-hierarchical approach to multimodeling is provided by Metropolis [16] and Colif [10]. This approach does not segregate distinct models of computation hierarchically.

Ptolemy Classic [9] also illustrates 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 [24], and has been developed into design tools like Metropolis [16]. Multi-view modeling has also formed a centerpiece of model-integrated computing [32] and has been applied in a number of large-scale system designs [17]. Multi-view modeling in the sense of providing alternative views of the same system, e. g. dynamically created during a simulation, is investigated by the KIEL system [31].

A third form of multimodeling, where a single model can be specialized to multiple distinct implementations [29], 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 [14]. Our approach was also influenced by Huang [21]. We also leverage actor-oriented classes [26, 23, 25] to maintain consistency across multi-view models.

3 The Traffic Control Model

A Statecharts model [19] of a simple traffic light controller is shown in Fig. 1. The module TRAFFIC_LIGHT has two states. The left state (the initial state) 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,

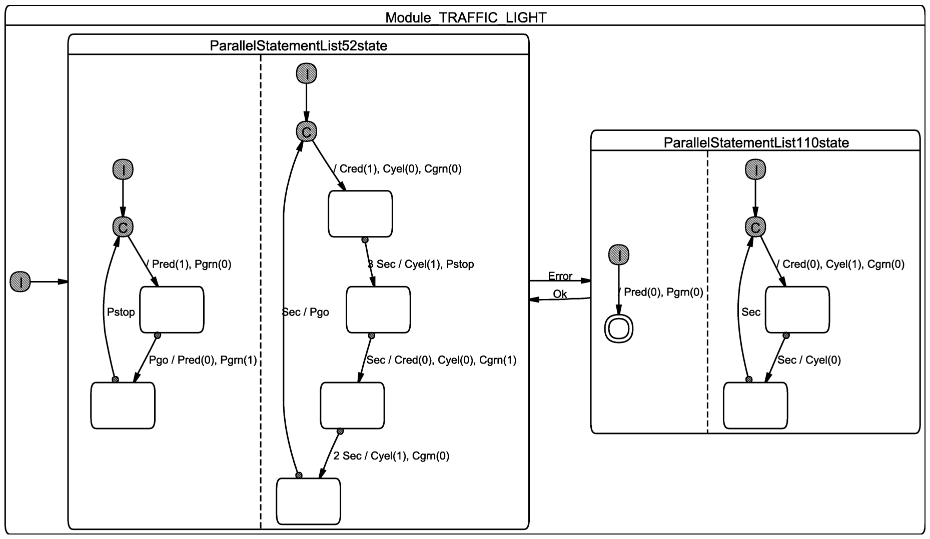

Fig. 1. A Statecharts model of a simple traffic light controller.

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 an action.

The concurrent car control state machine is to the right of the dashed line that separates it from the pedestrian light control state machine. It 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 [8]. 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 [19].

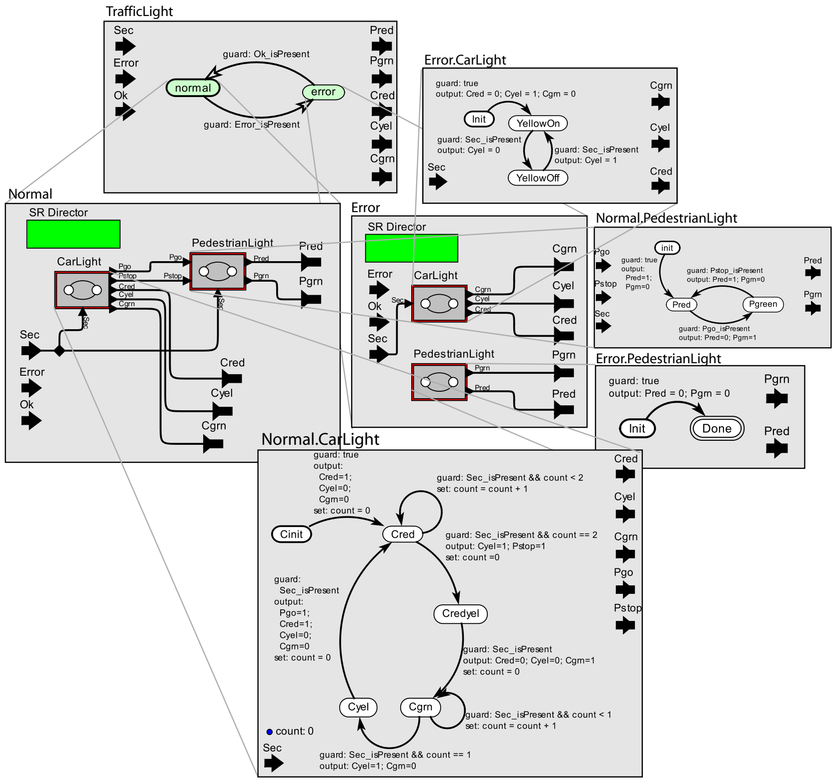

Fig. 2. The traffic light controller in Ptolemy II.

In this paper, we use Ptolemy II [13] as a laboratory for experimenting with model engineering precisely because it embraces all the forms of multimodeling in which we are interested. In Ptolemy II, the hierarchical combination of MoCs is more explicit than in Statecharts. A Ptolemy II model equivalent to that of Fig. 1 is shown in Fig. 2. The top level of the hierarchy, labeled TrafficLight, just like Fig. 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 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** [15] to distinguish them from classical state machines. This mirrors 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 that when a modal model fires (which in the SR MoC occurs on each tick of a global clock [6]), the refinement of the current state is fired, 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, 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 Fig. 1. Unlike Fig. 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 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 Fig. 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 Fig. 1, it is very hard to tell whether there is feedback between the "and states." That is, it is hard to tell whether one 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 Fig. 2, but not in Fig. 1. Note that both SCADE and Esterel-Studio also support such an explicit syntax [7]. In Fig. 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 Fig. 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 Fig. 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.[4]

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. 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 Fig. 2 is the **final state** in Error.PedestrianLight, which is shown with a double ellipse. This is similar to the double circle in Fig. 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 Fig. 2 is larger (more verbose). However, it is also

[4] Since guards and transitions involve arbitrary operations on variables, the question of whether a modal model is determinate is undecidable in general. However, it is feasible to perform static checks at least for a decidable subset, e.g. by interfacing the modeling tool to a theorem prover [20].

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 hierarchy, the diagram presents a simple abstracted view of the system. Such an explicitly hierarchical syntax may scale more easily to large models. Moreover, we will show below that it facilitates sharing of model components across distinct models. Note that although the model in Fig. 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 Fig. 1 has been synthesized from an equivalent, textual Esterel description [30], and it seems feasible to alternatively synthesize the model in Fig. 2.

5 Modeling the Environment

The top levels in 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 rich enough to correctly 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 [27] 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 Fig. 3. The TrafficLight actor is the modal model shown in Fig. 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.

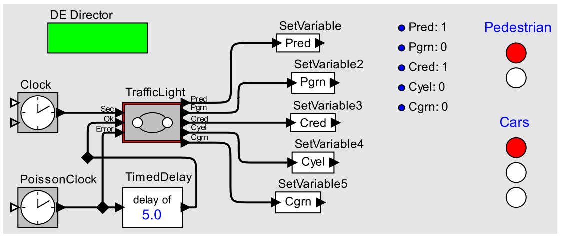

Fig. 3. A DE model of the traffic light environment in Ptolemy II.

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 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.[5] 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 Fig. 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 is shown in figures 4 and 5. The top level of this model uses the WirelessDirector, a generalization of the DE Director that supports wireless communication models [4]. 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

[5] 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.

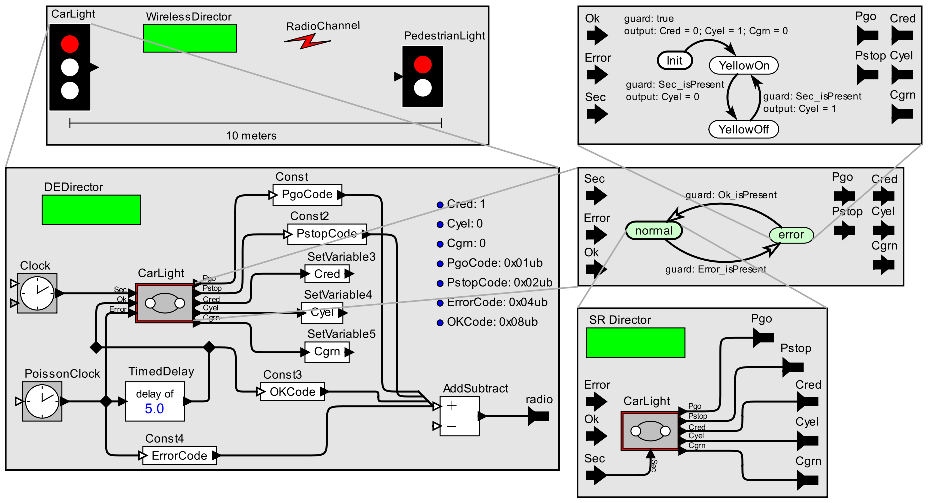

Fig. 4. Wireless deployment, showing the car light model.

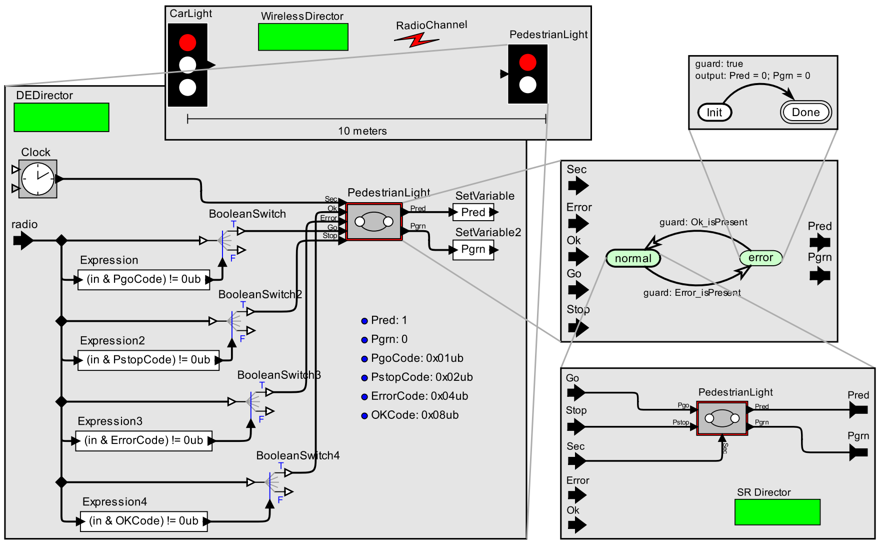

Fig. 5. Wireless deployment, showing the pedestrian light model.

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 icons constructed by the model engineer that contain circles that change color in response to changes in the values of their parameters. The output port of the CarLight actor communicates wirelessly via 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 Fig. 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 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 an important aspect of the design, absent from 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 not a desirable property of the system.

A designer faces two choices: she could refactor the functional model in Fig. 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 new model for the deployment. Here we do the latter. Consequently 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.

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?

Our model uses instances of two actor-oriented classes [26] to accomplish this objective. The CarLight and PedestrianLight actors in figures 4 and 5 are instances of the same model definitions labeled in the Fig. Normal.CarLight and Normal.PedestrianLight. 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 Design for Verifiability

Models for safety critical systems must be verified. In our traffic light example, we may want to prove the impossibility for the car light and the pedestrian light to be green simultaneously. The functional and deployment models given above are not immediately suitable for formal verification.

The standard approach at this point would be to construct by hand a third model suitable for verification. This would be an FSM model against which a temporal logic specification could be checked via model checking [12]. Our approach is to synthesize from the functional model Kripke structures specifying the behavior of the system, and checking them against a CTL specification of a safety property. For our traffic light, the following formula is suitable:

```
! EF (CarLightNormal.state = Cgrn
     & PedestrianLightNormal.state = Pgreen)
```

This can be read, "there does not exist a future where the car light and the pedestrian light are both green." We automatically generate an SMV model of the traffic light, combine it with the above (manually specified) safety property, and use NuSMV [11] to prove that the property is satisfied. The synthesis of the SMV model leverages the code generation infrastructure in Ptolemy II [33].

Our models include FSMs, suggesting that translation to SMV would be relatively straightforward. Examination of Fig. 2 reveals a number of challenges, however. First, we have built the models using particular concurrent MoCs the semantics of which must be accurately represented in an SMV model. Specifically, the SR and DE semantics and their interaction with FSMs must be carefully reflected in the SMV model, to the extent possible, and abstracted when they cannot be exactly reflected. Second, any two FSM modeling tools will inevitably have subtle semantic differences. Some can have profound consequences for verification, such as the support for so-called "history transitions" in hierarchical FSMs. Third, the FSMs in Fig. 2 are actually *extended FSMs*, since they include operations on the count variable. We apply abstract interpretation to encode values of such inner variables into atomic propositions. We use techniques inspired by region automata [1] to generate a compact variable domain.

An outline of the .smv model, including the CTL safety property, is given in Fig. 6. This model is automatically generated from the model in the "normal" state of Fig. 2 using code generation techniques, and hence is automatically maintained along with the functional model.

NuSMV verifies that our safety property is satisfied by this model, but this exercise reveals a pitfall of multimodeling. The upper CarLight component in the refactored deployment model of Fig. 4 is not the same as the one that we verified in Fig. 2. In fact, the deployment model does not satisfy the safety property. Even though we used actor-oriented classes to share definitions across models, we did not (and could not) apply the safety check to the portion of the model that is shared. The models require further refactoring (and some further design) to ensure that safety conditions verified on one model also apply to another. We consider it an open challenge to provide model engineering techniques that mitigate such pitfalls.

8 Conclusion

Analogous to software engineering as a discipline, we focus on problems that arise in *model engineering*. We show that distinct models constructed in different ways can be usefully combined to form new models (*hierarchical multimodeling*), and that distinct models constructed for different purposes can share key parts of the design (*multiview modeling*).

Specifically, 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 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 given a deployment model, which defines an implementation using wireless communication 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 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) is 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.

Finally, we show that a third model constructed for the sole purpose of verification can be synthesized using code generation techniques from the functional model. This model can be used prove a safety property using model checking. However, since the deployment model fails to satisfy the same safety property, this exercise could result in false confidence in the model. How to mitigate this risk remains an open challenge.

We have considered multiple models of the same system to separate functional aspects from deployment and verification. The designer should be able to specify different aspects of the same system independently, to allow a clean separation of concerns while keeping a model consistent. However, multi-view modeling can be applied at different levels and in very different ways. For example, it can refer to the animation of a model during a simulation, or to the alternation between graphical and textual representations, or indeed also to the alternation between a monolithic Statechart model and an explicitly hierarchical syntax, as discussed in this paper. At this point, it is still largely up to the modeler to construct different views of the same system. How best to harness a modeling system to assist the user with this task still seems to be a largely open problem.

9 Acknowledgments

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

References

1. R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
2. C. André. SyncCharts: A visual representation of reactive behaviors. Technical Report RR 95–52, rev. RR (96–56), I3S, April 1996.
3. C. André. Semantics of S.S.M (Safe State Machine). Technical report, Esterel Technologies, April 2003.
4. 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.
5. 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 *LNCS*, pages 128–148, Lübeck, Germany, September 19-23 1994. Springer-Verlag.
6. A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, 1991.
7. G. Berry. The effectiveness of synchronous languages for the development of safety-critical systems. White paper, Esterel Technologies, 2003.
8. G. Berry and G. Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.
9. 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.
10. 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.
11. A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Computer-Aided Verification (CAV)*, volume LNCS 2404, page 359364, Copenhagen, Denmark, 2002. Springer.
12. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
13. 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.
14. 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.
15. 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.
16. G. Goessler and A. Sangiovanni-Vincentelli. Compositional modeling in Metropolis. In *Second International Workshop on Embedded Software (EMSOFT)*, Grenoble, France, October 7-9 2002. Springer-Verlag.

17. 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.
18. C. Hardebolle and F. Boulanger. Modhel'x: A component-oriented approach to multi- formalism modeling, October 2 2007.
19. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
20. M. P. E. Heimdahl and B. J. Czerny. Using PVS to analyze hierarchical state-based requirements for completeness and consistency. In *High-Assurance Systems Engineering Workshop (HASE)*, page 252, Washington, DC, USA, 1996. IEEE Computer Society.
21. Y.-S. Huang. Design of traffic light control systems using Statecharts. *The Computer Journal*, 49(6):634–649, 2006.
22. 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.
23. G. Karsai, M. Maroti, . 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.
24. 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.
25. 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.
26. 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.
27. E. A. Lee and H. Zheng. Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In *EMSOFT*, Salzburg, Austria, October 2007. ACM.
28. 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.
29. 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.
30. S. Prochnow, C. Traulsen, and R. v. Hanxleden. Synthesizing safe state machines from Esterel. In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Ottawa, Canada, June 2006.
31. S. Prochnow and R. von Hanxleden. Statechart development beyond WYSIWYG. In *International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, Nashville, TN, USA, October 2007. ACM/IEEE.
32. 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.
33. G. Zhou, M. Leung, and E. A. Lee. A code generation framework for actor-oriented models with partial evaluation. Technical Report UCB/EECS-2007-29, University of California, EECS Department, February 23 2007.

```
MODULE CarLightNormal( Sec_isPresent )
    VAR
        state : {Cyel,Credyel,Cred,Cinit,Cgrn};
        count : { ls,0,1,2,gt };
    ASSIGN
        init(state) := Cinit;
        next(state) :=
            case
                state=Cinit & count=ls :{ Cred };
                ...
                Sec_isPresent & state=Cred
                              & count=ls :{ Cred };
                ...
                1             : state;
            esac;
        init(count) := 0;
        next(count) :=
            case
                state=Cinit & count=ls :{ 0 };
                ...
                Sec_isPresent & state=Cred
                              & count=ls :{ ls };
                ...
                1             : count;
            esac;
    DEFINE
        Pstop_isPresent :=  ( Sec_isPresent
                              & state=Cred & count=2 ) ;
        Cred_isPresent :=  ...
        Cgrn_isPresent :=  ...
        Pgo_isPresent :=  ...
        Cyel_isPresent :=  ...
 MODULE PedestrianLightNormal(Pstop_isPresent,Pgo_isPresent)
    VAR
        state : {Pinit,Pgreen,Pred};
    ASSIGN
        init(state) := Pinit;
        next(state) :=
            case
                state=Pinit :{ Pred };
                Pgo_isPresent & state=Pred :{ Pgreen };
                Pstop_isPresent & state=Pgreen :{ Pgreen };
                1             : state;
            esac;
    DEFINE
        Pred_isPresent :=  ...
        Pgrn_isPresent :=  ...
 MODULE main
    VAR
        CarLightNormal: CarLightNormal( 1);
        PedestrianLightNormal: PedestrianLightNormal(
          CarLightNormal.Pstop_isPresent,
          CarLightNormal.Pgo_isPresent );
    SPEC
        ! EF (CarLightNormal.state = Cgrn
             & PedestrianLightNormal.state = Pgreen)
```

Fig. 6. The SMV model generated from the “normal” state refinement in Fig. 2.