Using Separation of Concerns for Embedded Systems Design [*]

Ethan K. Jackson
Institute for Software Integrated Systems
Vanderbilt University
Box 1829, Station B
Nashville, TN 37235
ejackson@isis.vanderbilt.edu

Janos Sztipanovits
Institute for Software Integrated Systems
Vanderbilt University
Box 1829, Station B
Nashville, TN 37235
janos.sztipanovits@vanderbilt.edu

ABSTRACT

Embedded systems are commonly abstracted as collections of interacting components. This perspective has lead to the insight that component behaviors can be defined separately from admissible component interactions. We show that this separation of concerns does not imply that component behaviors can be defined in *isolation* from their envisioned interaction models. We argue that a type of behavior/interaction co-design must be employed to successfully leverage the separation of these concerns. We present formal techniques for accomplishing this co-design and describe tools that implement these formalisms.

Categories and Subject Descriptors

D.2.2 [**Software**]: Software Engineering—*Design Tools and Techniques*; D.3.1 [**Software**]: Programming Languages—*Formal Definitions and Theory*

General Terms

Design, Languages, Theory

Keywords

Models of Computation, Embedded Systems, Multiple-Aspect Modeling, Separation of Concerns

1. INTRODUCTION

Embedded systems theory lacks a formal compositional semantics that is both sufficiently general and analyzable. Researchers coped with this absence by defining many less general, but nonetheless analyzable, semantics. The goal then became one of stitching back together less general semantics to create broader semantic domains that remained amenable to analysis. A fundamental insight was that a semantics can be divided into two parts: component behavior and component interaction. This *separation of concerns* made it possible to hierarchically compose semantics by defining the internal behavior of a component with a different semantics than the interaction model.

Separating behavior from interaction has been essential in reconciling the disparity between semantics, but it may lead embedded system designers to make a wrong conclusion: Intended component behaviors can be designed in isolation from their intended interaction models. We show that this is not true with a simple proof by contradiction. Consider the example in Figure 1. This function contains a typical

```
1. T Transform (T obj, String name) {
2.     if (obj != NULL)
3.         return ( P(obj) ?  TR(obj) :  obj );
4.     else if (name != NULL)
5.         return ( P(Get(name)) ?  TR(Get(name))
6.                              :  Get(name) );
7.     else return NULL;
8. }
```

Figure 1: Function applies a transformation `TR()` to an object if the property `P()` holds.

sequence of operations: Check an object for a property `P`, and if `P` holds then transform the object with `TR` and return the transformed object. If the property `P` does not hold then return the untransformed object (lines 3, 5 - 6). We also add a primitive type of polymorphism to this function. If `Transform` is passed a non-null object of type `T` through the `obj` parameter, then apply the sequence to `obj` (line 2). Alternatively, if `Transform` is passed a string identifier through the `name` parameter, then look up the object called `name` and apply the operations to this object (line 4).

A developer might consider `Transform` to be a useful component, and attempt to use it under various interaction models. However, even this simple component cannot be sensibly used under some interaction models, and this contradicts that component behaviors can be constructed in isolation from interaction models.

Let the `Transform` component be a unit of black-box behavior, and use it inside a synchronous programming language [3],[13] like SIGNAL [14]. In the notation of SIGNAL, we might say `out := Transform(in1,in2)`, meaning that the output signal `out` is the result of applying `Transform` to the input signals `in1` and `in2`. This construct implies that both `in1` and `in2` are synchronized, i.e. their clocks

[*]This research was supported by NSF grant CCR-0225610.

are the same ($\hat{\texttt{in1}} = \hat{\texttt{in2}}$). Therefore the only use-case allowed is that which calls `Transform` with both a reference and a name, but the intended use was to pass `Transform` either a reference or a name. It may be argued that the `NULL` reference is a value, so the streams should be synchronized, but testing for `NULL` carries the same meaning as testing for absence and `NULL` has no meaning as an object. The problem is that the component behavior uses a semantic construct (testing for absence) that is also provided by the interaction model. If we do not carry this semantic construct up into the interaction level, then the semantics of interaction may interfere with the intended component behavior. We correct this in the SIGNAL case by creating a correlation between an absent clock and the `NULL` value:

```
out :=(Transform(in1,NULL) when in1) default
      (Transform(NULL,in2) when in2).
```

Testing for absence is not a boundary case that interferes with all interaction models. For example, consider the `Transform` component in a time-triggered language like Giotto [15]. For our purposes this interaction model has two interesting properties: First, data is read and written through global buffers of size 1. Second, components are executed periodically, and they read data from global buffers at the beginning of the period and write data at the end of the period. In this environment the `Transform` component works as expected. When the component is activated it will read from the buffers `in1` and `in2`, and the data in these buffers may or may not be `NULL`. The component may or may not write an object to the output buffer `out`. This works without confusion because the Giotto interaction model does not define a notion of absence, so this environment does not interfere with the component's notion of absence.

These examples show that though parts of an embedded system can be independently specified at the behavior and interaction layers, the combined effect of the layering may not produce the intended results. Trickier still, the unintended behavior in the previous example is not technically an error behavior, so techniques like composition of interface automata would not have detected this mistake.

Problems like these also occur when the interaction model is decomposed into untimed and timed pieces. A similar claim has been made that the topology of a system and the time model used to schedule a topology are *orthogonal* concerns that can be separated. However, typically the same system topology cannot be arbitrarily reused across varying time models. At the very least, some static analysis must be performed on the untimed topology to decide if some schedule exists for the system. This also suggests that the term *orthogonal* is not accurate for these concerns. We argue that, in general, separate concerns like behavior and interaction, topology and scheduling, are not orthogonal, i.e. these pieces of a system should not be constructed in isolation from each other.

In this paper we reexamine the embedded system design flow assuming the nonorthogonality of a set of concerns that are traditionally considered to be orthogonal (behavior, topology, time/scheduling). Our fundamental result is that the interdependencies between nonorthogonal concerns can be managed by reasonable changes to design abstractions and tools. In Section 2 we describe our general methodology for handling interdependent concerns. Section 3 relates our work to previous work on MoCs and separation of concerns. Section 4 shows a non-trivial application of our approach to the synchronous reactive MoC. Section 5 discusses simulators and modeling environments built from MoCs with interdependent concerns. Finally, Section 6 describes our conclusions and future work.

2. APPROACH

First and foremost we wish to incorporate our assumption of nonorthogonal concerns into the design abstractions used for embedded systems design (MoCs) so that:

1. there exists some amount of "independent" reasoning within each concern,
2. the interdependencies between concerns are explicit,
3. interference of semantic constructs is detectable and hence preventable.

The first requirement preserves the notation of "separation". There should be some reasoning that can be done within a concern that has consequences only inside the scope of that concern. If this is not the case, then the concerns should not be separated. The second requirement builds the interdependencies into the MoC, so that they are explicitly part of the design abstractions. This allows the designer to explicitly reason about the interactions between concerns. The final requirement characterizes how semantic constructs are split across concerns. This requirement may disallow some systems that would have been valid had there not been any separation of concerns. As we will show later, this criteria would make the mistake in the previous example a detectable error.

Though there may not be a unique solution to these requirements, we propose a generic and novel abstract semantics that meets these requirements and can be specialized into several important MoCs. Our abstract semantics is partitioned into three layers, so that there is a layer for the component behavior, topology, and scheduling/time concern. The topology and time layers form the interaction model which describes how and when components communicate and fire. The topology layer is an *untimed* and *operational* interaction model [4] that schedules components and mediates data communication. This provides the developer with an untimed interpretation of the system and allows some amount of untimed intuition during design. There is a *timed* and *operational* [4] layer that modulates the untimed execution, and provides a timeline for events. The untimed execution of the system corresponds to a plant, and the timed execution corresponds to a controller that observes and modulates the execution of that plant. Both the untimed and timed layers may have restrictions on the systems that can be constructed at these layers, and each layer may impose constraints on the other. A particular model of computation (MoC) is specified by untimed and timed operational interaction models, the constraints on each layer, and the constraints that each layer imposes on the other.

This approach makes explicit two types of interdependencies. First, there is the "operational interdependency" that is the feedback loop between the timed "controller" and the untimed "plant". The timed system modulates the untimed system via a set of controllable actions, and the untimed system affects the timed system through variables that are observed by the controller. Second, there are the "constraint-based interdependencies" that are restrictions that the system at one layer can impose on the system at another layer.

We have incorporated nonorthogonal concerns into the MoCs, and so we can explore how this affects tools based on MoCs. Specifically, we examine the ramifications on simulators and modeling environments. Though many ramifications can be considered, we chose to investigate how interdependencies affect simulator construction and reuse. Our primary result is that we can construct an "abstract simulator" from our abstract semantics. This abstract simulator contains most of the necessary structure and implementation to simulate the three layers. It also implements the operational interdependencies so that each layer correctly influences the others. A simulator for a particular MoC is constructed by filling in some concrete details. This allows the implementation of the abstract simulator to be highly reusable, and may allow some reuse of concrete simulators. This shows that it is possible to construct simulators with little effort (by just adding the concrete details), even though the MoC is complicated by interdependent concerns.

At the modeling level we show that modeling tools can be used to mitigate the additional modeling complexity incurred by interdependent concerns. We use multi-aspect modeling [6] to associate an aspect with each concern. A system is then built concurrently in each aspect and this allows the co-design of the timed and untimed parts of the system, as well as the immediate observation of layer interactions. Multiple-aspect modeling can manage the constraint-based interdependencies by performing design-time checking of constraints in and across modeling aspects.

3. BACKGROUND

Our work relates to previous work on the heterogeneous composition of semantics, and to the design of MoC oriented tool architectures. We begin by discussing the former. Semantics are composed hierarchically by specifying an interaction model M under which a set of components $c_1, c_2, \ldots, c_n$ interact with each other. Inside a component c_i there maybe another network of components $c'_1, c'_2, \ldots, c'_n$ that interact according to a different interaction model M', and this hierarchy can be arbitrarily deep. Several important questions arise when composing interaction models hierarchically: Which interaction models can be composed? How should data cross component boundaries? How should time be coordinated across component boundaries? Which known properties are preserved when components are connected under interaction model M? Which new properties emerge when components are connected under M?

Different approaches have been used to decide if two MoCs can be composed. One approach is to define a non-restrictive means of connecting components so that components and their corresponding MoCs can be arbitrarily composed. Analysis of a particular component topology may then reveal if the system uses only valid compositions. Behavioral types and interface automata have been used for such analyses [11]. However, analysis can be skipped if the user is willing to risk a run-time failure of the system. This approach is general, but may allow invalid compositions that may or may not be detectable at design time. A second approach is to define a family of related MoCs using an abstract semantics. Formal composition operators can then be defined for any two members of the semantic family [9]. This approach is not as general as the first, but it provides formal reasoning about the meaning of compositions. Abstract semantics approaches of this sort can be easily incorporated into tools that support the first methodology.

Researchers have gravitated towards different layering approaches to answer different questions and to develop different tools. We now describe some representative approaches and their associated tools. Ptolemy II is (among many things) an exploration in heterogeneous modeling and simulation of embedded systems [10]. Components in Ptolemy II are called *actors*, and the actor topology is primarily a syntactic construct. Ptolemy II places all of the semantics of communication and timing into *directors*. A director is an operational construct that analyzes the actor topology to generate a schedule, and without a director the actor topology does not have a well-defined meaning.

The IF tool suite focuses on the representation and verification of embedded systems [12]. Formal reasoning is required so IF uses an abstract semantics that is built on top of timed-automata extended with dynamic priorities. IF separates a MoC into two parts: The *interaction model* is a denotational semantics that places constraints on how components can interact with each other. The *scheduler* is an operational semantics that coordinates the execution of timed automata . Unlike Ptolemy II, a component topology in IF has a well-defined meaning even without defining the scheduler.

Metropolis is a tool suite for platform-based design [1]. It provides techniques for mapping an abstracted (e.g. functional) description of a system onto a platform that is built from components with non-negotiable properties and characteristics (e.g. processor speeds, bounded buffers, a particular bus protocol, etc...). This goal requires Metropolis to define the interaction model, and this is done partially by the Metropolis Meta-model (e.g. processes, media), and partially by the user using constructs like Quantity Managers. Metropolis advocates the separation of concerns, but it does not fix the way in which concerns must be separated. We see Metropolis as a framework for implementing various layering strategies.

The separation of concerns is advocated in all of these methodologies, as is the view that these concerns are orthogonal: Plotemy II claims that "actor-oriented design orthogonalizes component definition and component composition, allowing them to be considered independently." [5] Similarly, in IF "the proposed composition distinguishes clearly between two different and orthogonal aspects of systems modeling: behavior and interaction (architecture)." [9] The Metropolis Meta-model makes similar claims that behavior and architecture "are fairly orthogonal." [8] We argue that these concerns are interdependent and we present a methodology for managing this interdependence for embedded systems design.

4. SYNCHRONOUS REACTIVE EXAMPLE

We introduce our work by reformulating the synchronous reactive (SR) MoC according to the layering that we previously described. This is a large non-trivial example, with some interesting ramifications on designing SR systems. First, we define an untimed operational semantics, $BFlow$, that schedules components in bounded memory dataflow networks. Second, we define a timed operational semantics, $DRModes$, that modulates the execution of the untimed plant so that determinism and reactivity are guaranteed.

Third, we describe the constraints in and across the layers, and show how these layers fit together to define the SR MoC. Finally, we show how this layering prevents interference of semantic constructs, and we show how it can be leveraged in tools. The result is a formulation of the SR MoC that meets the requirements for the separation of nonorthogonal concerns.

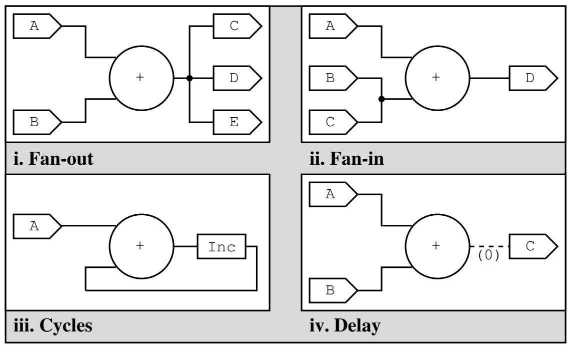
4.1 Untimed Dataflow with *BFlow*

Some synchronous programming languages (e.g. Lustre and Signal) describe computation in terms dataflow graphs. We will use this approach because there is a simple untimed operational semantics for dataflow graphs: An n-ary dataflow operator has n inputs, and if the operator observes a data token on each of these inputs, then it consumes these tokens and produces a token on its output. To be more precise, the untimed execution is a sequence of *runs.* Figure 2.a describes what can occur during a run, and how a run ends. The first three properties are common, while properties 4-6 generalize the notion that tokens are conserved. Instead of requiring that every created token must be consumed (i.e. conservation of tokens), we require that every token waiting at the input of an operator must be consumed. Any token that is not waiting at an input is irrelevant and will be discarded at the end of a run.

In addition to the properties of a run, we allow additional features in the dataflow graphs, as summarized by Figure 2.b (To avoid excessive arrows, let the inputs be to the left of an operator and the outputs to the right.) The fan-out feature (2.b.i) means that an output token can be replicated and passed down additional connections. The fan-in feature (2.b.ii) means that several tokens can simultaneously arrive on different connections, where they will be merged into one connection. The merge procedure non-deterministically chooses a token to be passed along, and consumes the remaining tokens. Merging must be applied in accordance to the *write once* rule. For example, in Figure 2.b.ii, if a token from B had already been passed to the adder, then any token from C would have to be consumed, but not forwarded, so as to not pass two different tokens the second adder input. The cycle feature (2.b.iii) means that operators can sit inactive because they are in an unbreakable dependency cycle. Cycles do not contradict any properties of a run, unless they receive tokens from operators that are not in a cycle. In this case, they cannot consume these tokens. This problem occurs in Figure 2.b.iii because the adder can receive a token from the A operator, but it cannot consume this token. Finally, the delay feature (2.b.iv) allows connections to have state. Connections drawn with dotted lines transmit tokens that they have observed in previous runs. Delayed connections must have an initial value that can be transmitted on the first run. A delay connection consumes the token that is written to it, but transmits a token that it has previously observed, and these two actions occur independently. Therefore, a cycle with a delay connection is not really a cycle.

```
1. Fire Once:  An operator can fire at most once per
   run.
2. Eventually Fire:  An operator that can fire will
   eventually fire.
3. Write Once:  A connection can carry at most one token
   per run.
4. Consumption:  A token waiting at an input, must be
   consumed during the run in which it was produced.
5. End:  A run ends after neither (2) nor (4) can occur.
6. Reset:  After a run ends, all unconsumed tokens are
   discarded.
```

(a)



(b)

Figure 2: (a) Properties of a run (b) Features of *BFlow*

These properties and features imply that a dataflow graph is malformed if there exists a dataflow operator o such that only some of the inputs of o will receive tokens. Conversely, well-formedness means that if some operator p writes to input i_{o_k} of operator o, then all inputs $i_{o_1}, i_{o_2}, \ldots, i_{o_n}$ of operator o must be written by operators that can be scheduled. If this is not the case for every operator o, then the dataflow graph is malformed. Though not implied by the execution rules, an operator with dangling inputs is consider malformed, because such an operator can never be scheduled.

In order to complete the operational definition of *BFlow*, we must describe how a controller can manipulate the untimed execution. First, a controller must be able to observe whether a run has ended, and it must be able to "peek" at the vector of tokens waiting at the external inputs of the untimed system. We will call these operations `RunEnded` and `Peek`. The controller must be able to start a run, and to clear off state information that accumulated from previous runs. The `RunStart` and `Clear` operations accomplish these tasks. Note the behaviors of `Peek`, `RunStart`, and `Clear` are not defined during a run. All of these operations are typical, but we now add one atypical action that a controller can take. A controller can *remove* an operator from a dataflow graph. If the `Remove` action is performed on a dataflow operator before a run is started, then the run proceeds as if that operator were never in the dataflow graph. (Note that the connections around the removed operator remain in the graph.) Of course, a `Remove` operation may result in a malformed dataflow graph. Therefore, a `Remove` operation is valid only if the resulting graph is well-formed. Finally, if a `Remove` operation is called during a run, then its behavior is undefined.

The untimed semantics we have presented here resembles an extended form of synchronous data flow (SDF) (which should not be confused with the synchronous reactive MoC).

It is important to observe that $BFlow$ contains many features that are not allowed in the SR MoC. First, the merging feature is non-deterministic and this type of non-determinism is not permitted by the SR MoC. Second, there may be deadlock due to zero-delay cycles, and this contradicts reactivity. The major exceptional property of $BFlow$ is that all well-formed dataflow graphs use bounded memory, and the memory requirements can be decided statically. (We call it $BFlow$ to emphasize the boundedness of memory.) This is guaranteed by the consumption requirement, which ensures that all relevant tokens will be consumed, and allows irrelevant tokens to be discarded.

4.2 Timed Controllers with $DRModes$

Modes capture the notion that a system has multiple and distinct global behaviors. A single mode is a configuration of the entire system, and when the system switches from one mode to another it reconfigures itself according to the new mode [7]. After a system is configured, it evolves according to that configuration. We consider a controller to be an entity that reconfigures the untimed dataflow graph. A controller, which we call a *structurally constrained modal model* (SCMM), is specified by a set of modes, along with a set of criteria for deciding which mode to apply in given situation. Our modes differ from other work on modal models, because our modes cannot arbitrarily reconfigure the system; i.e. their structure is constrained.

A SCMM is a set of system configurations, and the operational semantics for a SCMM explains how to select the current mode, and how to reconfigure the system according to this mode. In order to make the discussion concrete, we will use an untimed system that computes $C = A \cdot \overline{B}$. However, the operational semantics of this untimed system was described in the previous section. A SCMM does not know how to execute an untimed plant, but only knows how to reconfigure a plant according to modes. The interplay between untimed execution, and system reconfiguration will be used to create the SR MoC.

Figure 3.a shows a set of modes that can be applied to the untimed component $C = A \cdot \overline{B}$ (Figure 3.b.1). Each mode is a graph that has the same topology as the underlying untimed dataflow graph. The dataflow graph structurally constrains the modes by placing requirements on mode topology. (This will be fully explored in the next section.) Notice that each mode has three types of entities: There are inputs, shown to the left of a mode, rectangular elements in the center of the mode, and an output at the right end of a mode. Each of these has a special meaning in the context of a mode, which is different from its meaning in the context of the untimed dataflow. The inputs represent criteria for applying a mode. If an input k in mode M is colored solid white, then mode M cannot be applied unless input k is declared to have some value (e.g. a token must be waiting at that input in the untimed dataflow). Conversely, if k is colored solid black, then mode M cannot be applied unless input k is declared to not have any value. For example, in 3.a.1, both inputs are white, which means that Mode 1 cannot be activated unless both A and B have external tokens in the corresponding untimed dataflow graph. Similarly, Mode 3 in 3.a.3 requires input A to be absent and input B to be present.

If a mode can be applied, then its application reconfigures the corresponding untimed dataflow graph. The rectangular elements correspond to computational elements in the underlying dataflow graph. If an element is black, then its matching untimed component is removed from the graph when the mode is applied. Once a mode is applied, the same removal rules hold true for inputs and outputs. Figure 3.b shows how each mode reconfigures the underlying dataflow graph. Note that output elements have special meaning when hierarchically composing SCMMs, but for the sake of brevity, we will not discuss hierarchical composition in this paper. Modes also have edges, however these edges do not represent the flow of data, but rather describe dependency on data. Thus, there are edges that indicate a data dependency that must be resolved during the application of the mode. These are drawn with a solid line, and correspond to non-delay edges in the untimed component. There are also edges that indicate a data dependency that was resolved in a previous application of some other mode. These are drawn with a dashed line, and correspond to delay edges.

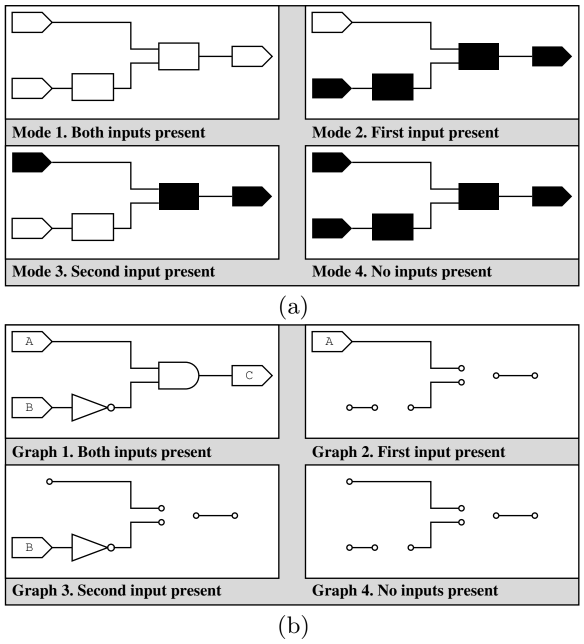

Mode 1. Both inputs present
Mode 2. First input present
Mode 3. Second input present
Mode 4. No inputs present
(a)
A
B
C
Graph 1. Both inputs present
Graph 2. First input present
Graph 3. Second input present
Graph 4. No inputs present
(b)

Figure 3: (a) Four modes and (b) the dataflow graphs induced by these modes.

The presence or absence of external tokens can effect which mode is applied to the untimed component. However, this mechanism alone is not sufficiently expressive to capture the SR MoC. We must also be able to select a mode based on the *data* contained in the tokens. It is sufficiently expressive to consider only boolean data, so to each input element k, we associate a boolean *input variable* i_k. Let $I = \{i_1, i_2, \ldots, i_n\}$ be the set of input variables, then we define a *mode entry constraint* to be a boolean function F over a subset I' of these variables. A mode with an entry constraint can be entered only if $F(I') = \texttt{true}$. If a mode M requires input element k to be absent, then i_k cannot be in I', because i_k is neither `true` nor `false`. If input element k is absent, then the associated input variable i_k has no value at all.

The SR MoC has several more features that must be added to this model. First, synchronous languages allow *constant data* to be used in the dataflow. In addition to an unchanging value, constants are special because their values can be present or absent, depending on context. Constants will be denoted as gray filled boxes. Second, many situations arise where the absence or presence of certain data is irrelevant to the mode. We call such requirements *don't care requirements*, and we denote them with an "X" written on top of the input element. If mode M has a don't care requirement on input element k, then mode M can be entered regardless of whether input k has data.

The untimed dataflow contains features that are not allowed by the SR MoC. We must utilize a controller to disable these features. The two features that must be disabled are the non-deterministic merge and the deadlocked zero-delay cycle. A non-deterministic merge occurs when multiple dataflow operators write to the same edge. We can disable this behavior by reconfiguring the dataflow graph to remove all but at most one of these operators. Similarly, a zero-delay cycle can be broken by removing one or more operators in the cycle, and the controller must do this. (We will formulate these rules more precisely in the next section.) Finally, we must say how the semantics provides a timeline for events. This is easy to do in the SR case. The controller examines the system inputs using a `Peek` operation and decides which mode to apply; this takes zero time. The controller reconfigures the dataflow graph using `Remove` operations; this take zero time. The controller indicates that the dataflow graph should be executed by using the `RunStart` operation. The controller waits until the `RunEnded` operation returns `true`. The time between when the run begins and ends is exactly one unit of logical time. This process repeats indefinitely, or until the system is reset by an external `Clear` operation.

4.3 Interdependencies

The previous two sections described the operational semantics of the untimed and timed systems, and described the operational interdependencies between these systems. We also eluded to some of the constraint-based interdependencies that act at and between these layers, and now we describe them all and discuss their ramifications. Potentially, each layer can constrain itself and the other layers, so there are n^2 possible sets of constraints, where n is the number of layers.

The behavioral layer is where the internals of a component are defined. Inside a component there may be C code, another SR system, or a system defined under some other MoC. Depending upon what is inside the component, it may be possible to formally project constraints up to the interaction layers. In another words, it may be possible to detect if an environment is interacting with a component incorrectly. While these constraints are important, they are largely orthogonal to this discussion. The example in the introduction interfaced with the `Transform` component correctly, but nonsensically. We do not use constraints to ensure adherence to a component's interface protocol, but to ensure that certain semantic constructs are only used at specific layers or not used at all. Therefore, we will not discuss constraints induced by the behavioral layer.

The untimed layer places interesting constraints on all three layers. The behavioral layer is constrained so that no dataflow operator outputs a `NULL` value. This forces component behaviors to use the notion of presence/absence that is put in place by the interaction model, regardless of the model of computation used inside the component.

$$\forall o \in \mathit{Operators}\ \neg(\texttt{Output}(o) = \texttt{NULL}) \tag{1}$$

The untimed layer also places constraints on itself. As discussed earlier, a dataflow graph must not have dangling inputs.

$$\forall o \in \mathit{Operators}\ (\texttt{NumConnected}(o) = \texttt{Arity}(o)) \tag{2}$$

where `NumConnected` counts the number of operator inputs that are connected to the output of at least one other dataflow operator. Additionally, if an operator will receive a token on one input, then it must receive a token on all inputs.

$$\begin{gathered}\forall o, o' \in \mathit{Operators}\ \exists p \in \mathit{Operators} \\ \texttt{CanSchedule}(p) \wedge \texttt{Connected}(p, o) \wedge \texttt{Connected}(o', o) \Rightarrow \\ \texttt{CanSchedule}(o')\end{gathered} \tag{3}$$

Finally, the untimed layer constrains modes in the timed layer. Given a dataflow graph d, all mode topologies must be isomorphic to d [1].

$$\begin{gathered}\exists \phi\ \forall m \in \mathit{Modes}\ (\phi(m) \cong d) \wedge \\ \forall i \in \texttt{Inputs}(m)\ (\phi(i) \in \texttt{Inputs}(d)) \wedge \\ \forall o \in \texttt{Outputs}(m)\ (\phi(o) \in \texttt{Outputs}(d))\end{gathered} \tag{4}$$

No mode can reconfigure a dataflow graph so that it violates constraint 2 or 3. This means that if an operator is present, all of its inputs must be present (or some may be constant)[2].

$$\begin{gathered}\forall o, o' \in \mathit{Operators}\ \texttt{DeclaredPresent}(o) \wedge \texttt{Connected}(o', o) \\ \Rightarrow \\ \texttt{DeclaredPresent}(o') \vee \texttt{DeclaredConstant}(o')\end{gathered} \tag{5}$$

The constraints that the timed layer places on itself ensure that non-determinism and zero-delay cycles are removed from the dataflow graph. The timed layer must disable all zero-delay loops.

$$\begin{matrix}\forall m \in \mathit{Modes} \\ \forall c \in \texttt{Cycles}(m)\end{matrix} \left\{ \begin{array}{l} (\exists v \in c\ \texttt{DeclaredAbsent}(v)) \vee \\ (\exists e \in c\ \texttt{IsDelayEdge}(\phi(e))) \end{array}\right. \tag{6}$$

Removing non-determinism means that if multiple operators write to the same edge, then at most one is declared present.

$$\begin{gathered}\forall m \in \mathit{Modes}\ \forall o, o' \in \texttt{Operators}(m)\ \forall e \in \texttt{Edges}(m) \\ \neg\texttt{DeclaredAbsent}(o) \wedge \texttt{WritesEdge}(o, e) \Rightarrow \\ \texttt{DeclaredAbsent}(o') \vee \neg\texttt{WritesEdge}(o', e)\end{gathered} \tag{7}$$

Finally, the controller cannot introduce any control non-determinism. This means that if more than one mode can be applied, then these modes must reconfigure the dataflow graph in the same way. Let the mapping ψ map from modes to dataflow graphs, so that $\psi(m, d)$ maps to the dataflow graph that is the configuration of d with mode m.

$$\begin{gathered}\forall s \in \mathit{Stimulus}\ \forall m, m' \in \mathit{Modes} \\ \texttt{CanApply}(s, m) \wedge \texttt{CanApply}(s, m') \Rightarrow \\ \psi(m, d) = \psi(m', d)\end{gathered} \tag{8}$$

The operational semantics provide constructs for system design and the constraints ensure that these constructs are

[1] Really dataflow graphs are multigraphs with other special properties, and we extend graph isomorphism to these objects without further discussion.

[2] This is a simplification of the full constraint. The complete constraint is an extended exercise in first-order logic.

partitioned into specific layers, or not use at all. (The constraints also enforce well-formedness rules inside a layer.) This partitioning exposes the intricacies of the MoC, provides the developer with an operational interpretation at every layer, and provides a formal means to enforce correct usage of the semantic constructs. Every feature discussed can be seen in the following example of a simple up-counter with asynchronous reset and non-volatile read (Figure 4). The reset signal `Rs` and the count signal `Cn` are boolean-

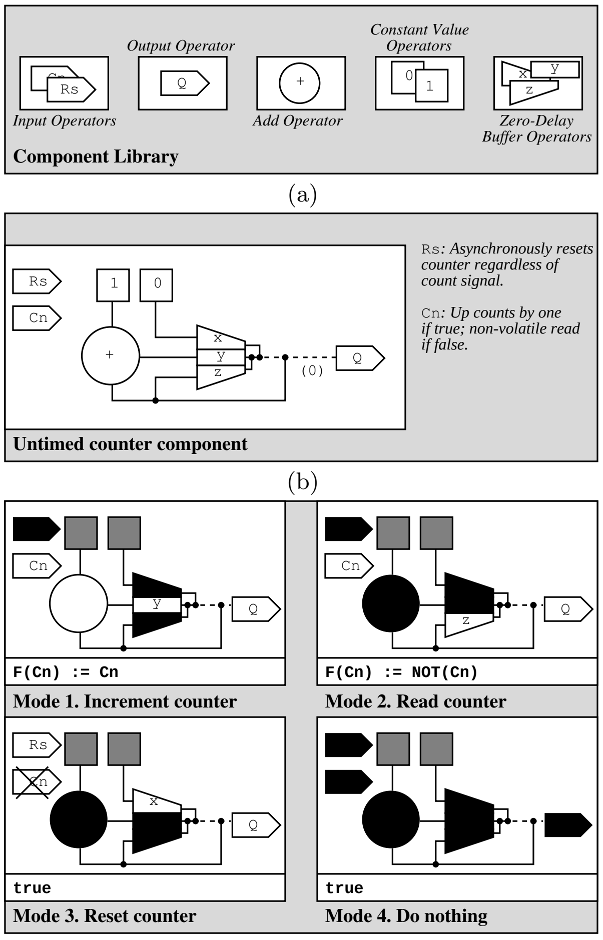


Figure 4: (a) A library of components (b) Components assembled into a counter (c) The modes that control the counter.

valued signals. If reset is present then it resets the counter regardless of the count signal. Assuming reset is absent, if count has a true value then the counter increments, otherwise the counter is read. Notice how semantic constructs are partitioned across the layers:

1. Behavioral layer: Data transformation (adder operator, constant value operator).

2. Untimed layer: Data movement (all edges in counter graph), Data merge (x-y-z merging), Intercomponent state (delay edge y-Q).

3. Timed layer: Presence/Absence testing (entry conditions on mode inputs), Data-dependent control flow (mode entry constraints in modes 1,2), Deterministic data movement (control of x,y,z components), logical time axis (one reaction per mode)

This suite of semantic constructs is not unique to our formulation of the SR MoC, but is available in one form or another in (almost) every synchronous language. This means that a user of such a language must keep track of all of these constructs and understand the way in which they interact. The intractability of causality analysis and global effects of local changes on the *clock calculus* does not make this any easier. Our approach provides semantic constructs through a layered set of operational semantics, and then uses explicit operational and constraint-based interdependencies to explain construct interactions and to enforce their correct usage. This provides the system designer with a clear and formal paradigm for MoC usage that can have tool support, as we will show in the next section.

Before we conclude our SR example, we should mention that previous work has tried to integrate the synchronous language Signal into an actor-oriented system like Ptolemy II [2]. This work proposed creating an actor for every Signal construct, so that these actors could run concurrently, albeit under some global scheduling mechanism, and receive tokens asynchronously. It was discovered that additional control signals were needed when converting constructs like `default`. By layering the semantics, we have implemented these constructs in an actor-like dataflow language (recall the operational definition of *BFlow*) without defining any special dataflow operators. In the up-counter example, the default operator was implemented by zero-delay buffers, and the additional control signals are a consistent part of our layering. These signals are not special case interactions specific to deterministic merge. In fact without any special dataflow operators we can handle every SR construct, including advanced features like time-based undersampling and oversampling. This is not completely obvious since we do not have space to fully describe all of the features of structurally constrained modal models.

5. TOOLS

We have presented a complex and well-known MoC to illustrate our approach. However, our work goes beyond the SR MoC in several ways. First, we have formalized our layering into a generalized actor-like dataflow semantics and a generalized SCMM controller semantics. The result is an abstract operational semantics that can be specialized for a particular MoC. *BFlow* and *DRModes* are the specializations needed for the SR MoC. Instead of presenting the algebraic formalization here, which is mainly of mathematical interest, we will present the applications of this formalization, which is of methodological interest.

Generalizing the operational interdependencies has a direct effect on simulator design. We have constructed a simulator called *TinyModes* that contains an implementation of our generalized abstract semantics. The component-based architecture of the TinyModes simulator is shown in Figure 5. The *Untimed* component contains the abstract operational semantics of the actor-like portion of the layering. Similarly, the *Timing* component contains the abstract operational semantics for the SCMM controller portion of the layering. The *Interaction* component coordinates communication between the simulators and corresponds to the for-

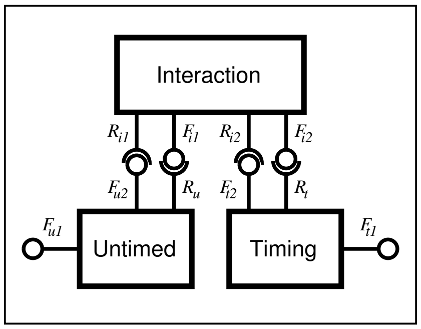


Figure 5: The Component-based architecture of *TinyModes* simulator

mal composition of the two layers. The simulator components interact through well-defined facets and receptacles. In order to construct a working simulator, these abstract components need to be specialized so that they implement a particular MoC. It is interesting to note that the actual C++ code for the abstract semantics uses abstract classes, and the act of specializing the semantics corresponds exactly to specializing abstract classes in the simulator code.

We have performed this specialization for the SR MoC, and so we can show the result of simulating the up-counter example with and without the SCMM controller. The gray

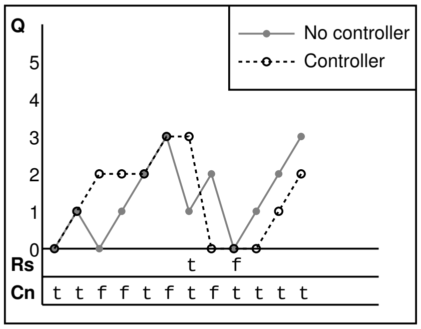


Figure 6: Simulation of up-counter with and without timed controller

line (Figure 6) shows the response of the counter when there is no controller. Notice how non-determinism results in a random application of either increment, read, or reset. This phenomenon can be thought of as plant behavior that must be controlled by the modal model. The dashed line shows the correct behavior of the counter when the controller is in place. Again, this type of reasoning is possible because of the layering, and the these two simulations are easily generated because of the simulator architecture. This architecture is also useful for simulator reuse. For example, we have implemented a time-triggered MoC, much like that implemented in Giotto, by reusing the same *BFlow* and attaching a different specialization of SCMM controllers.

Our use of constraint-based interdependencies to enforce partitioning of semantic constructs may seem like a burden to the developer. However, with multiple-aspect modeling the developer can rely on the modeling tool to either guarantee satisfaction of constraints via correct-by-construction approaches, or to find constraint violations and alert the user. The developer does not need to actively keep track of all the constraints, as this is managed by the tool, but the developer does need to understand the meaning of a violation and the correct way to fix a violation.

In the multiple-aspect modeling approach each layer becomes an *aspect*, which is a global view of the system from that layer. This means that there are two aspects: untimed, and timed. (There is no behavior aspect because the view inside a component is an entirely different from the system in which the component is used.) A system is modeled in both aspects concurrently and changes in one aspect dynamically affect the other aspect. This allows all the views of the system to attempt to make themselves consistent with modifications to a single view, or to report that they cannot be made consistent.

We have developed a generic modeling architecture for our abstract semantics using the multiple-aspect modeling and constraint checking facilities of the meta-programmable tool GME [6]. GME is a configurable modeling editor that uses a *meta-model* to configure itself for a particular modeling language. We have incorporated our abstract semantics into GME by constructing a meta-model that describes the modeling entities, aspects, and constraints necessary for our approach. This meta-model describes an abstract modeling language, so it cannot be directly used for modeling under a particular MoC. The abstract meta-model must be specialized for a particular MoC through specialization mechanisms like meta-model inheritance. This process is analogous to specializing the simulator components so that they implement a particular MoC. The result of this specialization is a ready-to-use multiple-aspect modeling tool with constraint management and correct-by-construction capabilities for layer co-design.

A particular realization of this abstract modeling language is *SMOLES* (*S*ynchronous *Mo*deling *L*anguage for *E*mbedded *S*ystems), which implements the SR MoC in GME. A part of the metamodel for untimed SMOLES is shown in Figure 7; it can be read much like a UML class diagram. This part of the SMOLES metamodel explains that a component contains exactly one dataflow graph and a number of input and output ports (*CmpInterfaceIn*, *CmpInterfaceOut*). A dataflow graph contains operators, and also has input and output ports that have an abstract superclass called *PortFCO*. GME uses UML stereotypes to explain how instances of a class will be visualized in the modeling language. The <<*Model*>> stereotype indicates that instances may contains other objects, and the <<*Atom*>> stereotype indicates that instances cannot contain other objects. Figure 8 shows the untimed dataflow graph for the counter example as it appears in the SMOLES/GME modeling environment. This figure is also the view of the component from the untimed aspect (or Dataflow Aspect as it is called in SMOLES).

The SMOLES metamodel also defines a timed language for SMOLES in the style of structurally constrained modal models. Figure 9 shows part of the metamodel for timed SMOLES, and it explains that a component contains a set of modes via an object called *ModeSet*. An individual mode can contain objects of type *ModeInput*, *ModeOperator*, and *ModeOutput*. These classes have stereotype <<*Reference*>> meaning that an instance points to some other object in the

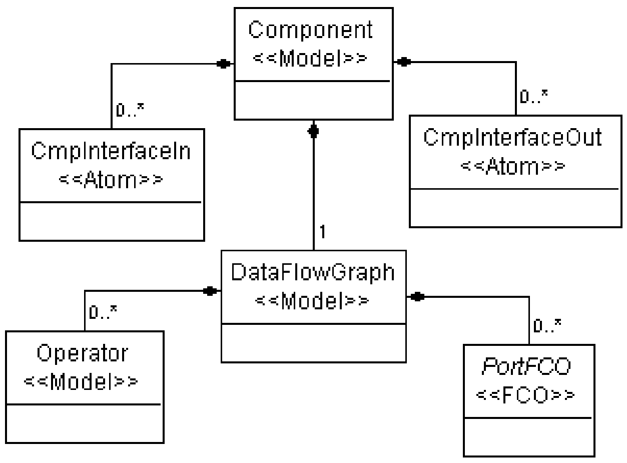


Figure 7: Selected part of the metamodel for untimed SMOLES.

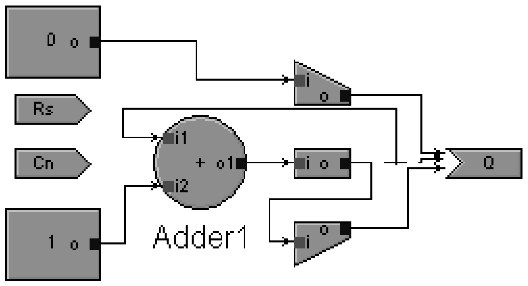


Figure 8: SMOLES/GME dataflow graph for the untimed aspect of the counter example.

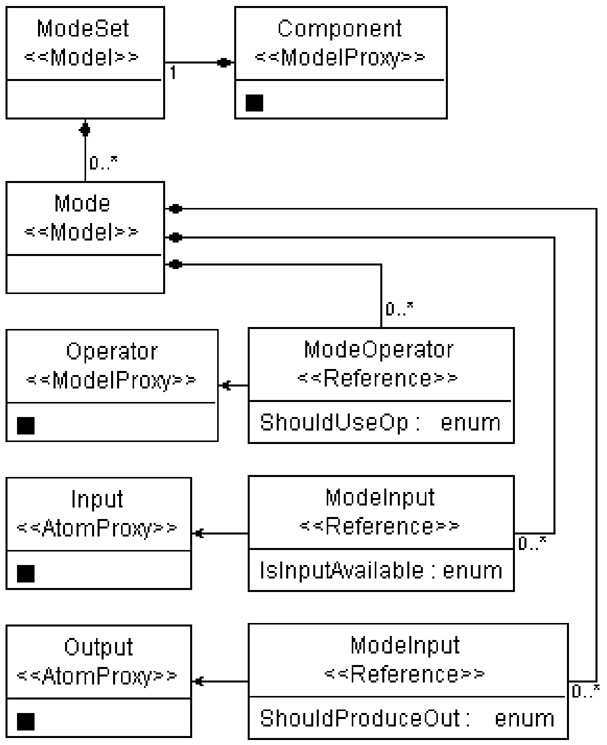


Figure 9: Selected part of the metamodel for timed SMOLES.

model. This reference mechanism maintains a one-to-one mapping (per mode) between objects in a mode and objects in the dataflow graph. Also notice that each instance of a mode object asserts whether or not its associated object should take place in the computation. For example the *ShouldUseOp* attribute asserts if the referenced dataflow operator should be used when the system is in that particular mode. This approach maintains a strict separation between untimed and timed information, and it allows both parts of the metamodel to be reused in the specification of other MoCs. Figure 10 shows the counter modes when viewing the counter system in the Mode Aspect. Note that dark gray indicates a *don't care* input and light gray indicates a constant value.

The SMOLES modeling environment manages MoC constraints using the facilities of GME. For example, mode topology must match the dataflow topology. This constraint is always correct by construction because GME automatically updates the Mode Aspect when changes in the Dataflow Aspect are made. During the construction of the graph in Figure 8, GME created the graphs in Figure 10. The only additional information the user had to supply was the absent/present information for each mode object. The active design-time maintenance of the aspects allows designers to codesign the untimed and timed parts of the system, while maintaining a conceptual separation between these two parts. Other constraints, like synchrony constraints, must be checked. GME will check constraints at a myriad of modeling events, and constraint violations have varying levels of severity that can range from a warning message to forcing an undo operation on the offending modeling action. For example, a simple synchrony constraint violation would occur if an operator asserted to be present received input from an operator asserted to be absent. This is captured in the SMOLES metamodel with the OCL constraint in Figure 11. This constraint appears formidable, but this is only because it must handle the details of model navigation. The logic is straightforward: If a mode object is present, then this implies that the input ports of the present object are not connected to the output ports of an absent object. The SMOLES metamodel contains a number of these constraints and this gives designers immediate feedback about the validity of their systems.

We believe that these tools illustrate the efficacy of our nonorthogonality assumption. If explicitly nonorthogonal approaches were shown to be too cumbersome to use, then the first order approximation of orthogonal concerns would be reasonable. However, we have shown that the additional complexity created by nonorthogonal concerns can be mitigated by multiple-aspect modeling approaches and interacting simulator architectures. The result of this nonorthogonal approach is a MoC that is carefully delineated into interacting parts, such that each part imposes clear constraints on the others. We believe that this approach to separated concerns will be more useful to designers because it produces a refined approximation of the design-time view of the system.

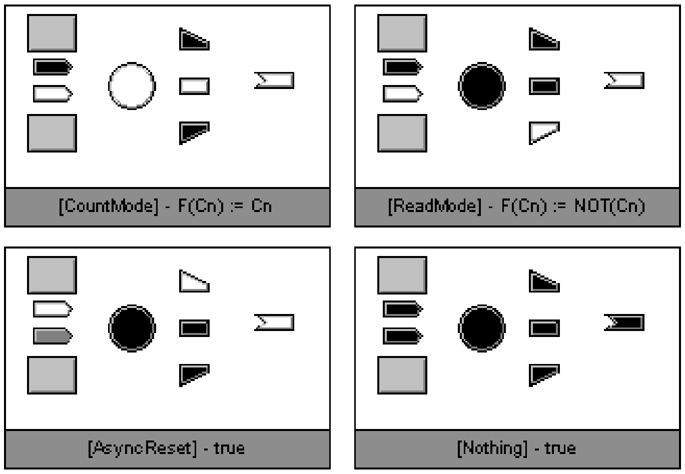


Figure 10: Counter modes in SMOLES/GME.

```
(self.ShouldUseOp = #Yes) implies
 self.parent().oclAsType(gme::Model).
  parts(ModeOperator)->forAll(o:ModeOperator|
   (o.ShouldUseOp <> #No) or
   (not(self.refersTo().oclAsType(gme::Model).
    parts(PrIn)->exists(pi:PrIn|pi.connectedFCOs("src",
     ModeExOutToExIn).intersection(
      o.refersTo().oclAsType(gme::Model).parts(PrOut))
       .size()>0))))
```

Figure 11: OCL constraint navigates model neighborhood to check for synchrony violations.

6. CONCLUSIONS AND FUTURE WORK

In conclusion, we provided evidence that component behavior, system topology, and scheduling concerns should not be considered as orthogonal or independent. We developed criteria for any MoC that considers these concerns to be nonorthogonal, and we presented a particular abstract semantics that meets these criteria. Our abstract semantics divides the interaction model into untimed and timed pieces, each of which is operational. We explored the consequences of nonorthogonal concerns on tools in the design flow. We showed that strategies based on abstract semantics can be used to build reusable simulators, and that multiple-aspect modeling can mitigate the additional complexities of nonorthogonal concerns. We conclude that the nonorthogonality assumption is a reasonable and useful refinement for separation of concerns, and that nonorthogonal concerns can be managed by design tools.

Future and on-going work will extend these principles to more models of computation. This work has driven new multiple-aspect modeling enhancements that can be expected in future versions of GME. The modularization of the simulator is a design pattern that will be reused in future “semantic libraries” that will come with GME. Finally, the relationship between constraint-based dependencies in aspects and the operational dependencies in simulators is also under study for further methodological interest.

7. ACKNOWLEDGMENTS

Special thanks to Albert Benveniste and Benoît Caillaud at IRISA for their invaluable input on this approach, and to the Columbus project for making this collaboration possible.

8. REFERENCES

[1] The metropolis meta model version 0.4. Tech. Rep. UCB/ERL M04/38, University of California, Berkeley, September 2004.

[2] A. Benveniste, P. Caspi, S. E. N. H.-P. L. G., and de Simone, R. The synchronous languages twelve years later. *Proceedings of the IEEE 91*, 1 (2003), 64–83.

[3] Boussinot, F., and de Simone, R. The esterel language. *Proceedings of the IEEE 79* (September 1991), 1293–1304.

[4] E. Lee, A. S.-V. A unified framework for comparing models of computation. *IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems 17*, 12 (December 1998), 1217–1229.

[5] E. Lee, S. N., and Wirthlin, M. J. Actor-oriented design of embedded hardware and software systems. *Journal of Circuits, Systems, and Computers 12*, 3 (2003), 231–260.

[6] G. Karsai, J. Sztipanovits, A. L. T. B. Model-integrated development of embedded software. *Proceedings of the IEEE 91*, 1 (January 2003), 145–164.

[7] G. Szedo, S. Neema, J. S., and Bapty, T. Reconfigurable target recognition system. *Proceedings of the FPGA Monterey, CA, Feburary 2000* (2000).

[8] G. Yang, Y. Watanabe, F. B. A. S.-V. Separation of concerns: Overhead in modeling and efficient simulation techniques. In *Fourth ACM International Conference on Embedded Software (EMSOFT'04)* (September 2004).

[9] Goessler, G., and Sifakis, J. Composition for component-based modeling. In *Proceedings of FMCO02* (November 2002), vol. 2852, LNCS, pp. 443–466.

[10] Lee, E. Overview of the ptolemy project. Tech. Rep. No. UCB/ERL M03/25, University of California, Berkeley, CA USA 94720, July 2003.

[11] Lee, E., and Xiong, Y. A behavioral type system and its application in ptolemy ii. *Formal Aspects of Computing Journal 16*, 3 (August 2004), 210–237.

[12] M. Bozga, S. Graf, I. O. I. O., and Sifakis, J. The if toolset. *Formal Methods for the Design of Real-Time Systems* (September 2004), 237–267. LNCS 3185.

[13] N. Halbwachs, P. Caspi, P. R., and Pilaud, D. The synchronous data flow programming language lustre. *Proceedings of the IEEE 79* (September 1991), 1305–1320.

[14] P. Le Guernic, T. Gautier, M. L. B., and Maire, C. L. Programming real-time applications with signal. *Proceedings of the IEEE 79* (September 1991), 1321–1336.

[15] T. A. Henzinger, C. M. Kirsch, M. A. S., and Pree, W. From control models to real-time code using giotto. *Control Systems Magazine 2*, 1 (2003), 50–64.