Correct-*ed* through Construction: A Model-based Approach to Embedded Systems Reality

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

Abstract

We present a design methodology for specifying embedded systems that addresses the complex nature of embedded systems design. Our approach uses modern model-based techniques to correct specifications as they are constructed, driving the engineer towards a more correct specification. We also present a concrete specification language based on this methodology.

1. Introduction

Embedded systems are difficult to design because of the resource constrained, real-time, and often safety critical nature of this class of systems. Any realistic design approach must allow the designer to specify rich and complex system dynamics (e.g. non-linear, time-varying behaviors), and to check that these dynamics satisfy system constraints. The problem is compounded by the varied spectrum of constraints that includes power requirements, deadline requirements, computational constraints, memory limitations, and security requirements. Furthermore, unlike traditional software applications, the correctness of an embedded system often depends on a good model of its environment. For example, the correctness of an autonomous flight controller depends on a good model for the dynamics of flight. This stack of requirements leaves us wondering if embedded systems design can, in general, be tackled.

Historical evidence seems to contradict this gloomy picture, at least a little. Historically, we have designed enormous computational systems, both in hardware and software, and the key to this success seems to be the design methodology. For example, the scalability of digital hardware design is due to the de facto design methodology that starts with HDL and RTL descriptions, and then synthesizes systems of boolean equations, and then maps these to cell libraries, and then synthesizes the layout [2]. Similarly, the scalability of traditional software can be attributed to the OOP methodologies that result in libraries of encapsulated behavior with low coupling, from which even bigger libraries of more complex behavior can be constructed [14]. Without these methodologies, we would not be able to manage the complexity of these enormous systems.

This discussion make us wonder: Will some design methodology save us from the complexity of embedded systems design? Before we discuss proposed methodologies that might suggest that the answer is **Yes**, let us point out that the answer may be **No**. As mentioned before, embedded systems design places a strong emphasis on global properties (deadlock freedom, power, etc...) and a correct embedded system usually means correct with respect to these properties. It may turn out that verification for essential classes of embedded systems is intractable, in which case we will have to restrict ourselves to small systems, or accept that certain properties are, for all practical purposes, unknowable.

Setting this negative possibility aside, there are already two major classes of design methodologies that have evolved, the first of which is the *verification* approach. In order to better contrast this approach with others, we call it the *Incorrect until Verified* (IuV) methodology. In the IuV methodology, one designs a system using the most convenient methodology, like OOP or logic synthesis, under the assumption that essential system properties may be incorrect and must be verified. During certain design increments the engineer passes the (partial) specification to a verification tool that checks global properties. If the properties are not satisfied, then a change must be made to the specification. In the worst case, this verification step is performed once, after the entire system has been specified. This approach has the advantage of flexibility: The engineer can design using any methodology without design restrictions as long as some verification tool can check the specifica-

tion. Of course, this caveat is the major problem with the IuV approach, because verification is intractable for many important properties so this approach does not scale in general. Additionally, for some classes of systems (e.g. some classes of hybrid systems) (sound and complete) verification tools do not exist.

The polar opposite of the IuV methodology is the *Correct by Construction* (CbC) methodology. The CbC methodology restricts the specification process so that only analyzable systems are specified. Some CbC approaches go so far as to make it impossible to specify an incorrect system with respect to certain properties. For example, in the time-triggered language Giotto [15] all specifications are inherently deadlock free (though not necessarily schedulable), so deadlock freedom does not need to be verified. Other CbC methodologies keep the specifications in a realm where analysis is tractable. For example, in synchronous dataflow (SDF) languages, schedulability is decidable in polynomial time [9]. Clearly the advantage of the CbC approach is that it is guaranteed to scale with respect to properties. The disadvantage is that no known CbC methodology can handle all interesting classes embedded systems. Thus, it is possible that a system of interest cannot be specified in a particular CbC tool, in which case one must find a different tool or resort to the IuV approach and hope that sufficient verification tools exist.

In this paper we present a new design methodology that handles the tricky reality of embedded systems design. Our methodology is scalable and has utility regardless of whether or not a grand unifying methodology for embedded systems design exists. We call our approach the *Corrected though Construction* (CtC) methodology. In the CtC methodology, the specification is *continuously* checked for errors using only polynomial time and space algorithms. In some situations the specification can be automatically corrected when problems are discovered. This approach cuts off the cycle of specification and verification that encounters intractability walls, allowing a specification to be continually improved. Simultaneously, the CtC methodology allows rich specification languages that aid the engineer in designing large scale systems. Of course, all these advantages do not come for free. By restricting ourselves to polynomial time and space algorithms, we cannot decide if properties are satisifed, however we can approximate them in a conservative sense: If a violation occurs, then the property will fail when verified, or, the system cannot be constructed under a particular CbC tool. If a violation does not occur, then the property may or may not be satisfied.

The CtC methodology complements the current methodologies by driving the designer towards better design decisions. In another words, by checking each design step as much as tractably possible, the CtC methodology drives the designer towards a specification that is more correct. By building the CtC methodology on top of model-based frameworks, we can translate the specification to a verification tool or tool a CbC tool, so we do not eliminate the use of these other methodologies. Simultaneously, if the universe is so malevolent that embedded systems design does not scale, then our approach is a reasonable one to building large scale systems that are more likely to be correct.

Our strategy for implementing a model-based CtC methodology is as follows: We begin by defining a specification language using a *metamodel*. This metamodel characterizes the structural semantics (syntax) of the specification language. Second, we *formally* connect this specification language to an operational semantics using *semantic anchoring* [8]. Third, using this formal anchor we find *conservative approximations* [13] to behavioral properties and encode them in the metamodel. Fourth, we find *generative actions* that can generate parts of a specification in order to correct problems. Finally, we write translators that can translate the specification to other verification or CbC tools.

This is a non-trivial process with many steps. But, given the complexity of embedded system design, the number of steps is not surprising. In this paper we illustrate a complete CtC specification tool called SMOLES^2 (Synchronous Modeling Language for Embedded Systems Specification) that is formally anchored to the synchronous reactive (SR) class of systems [1]. The SMOLES^2 language is rich with constructs like hierarchical composition and separation of concerns that were thoroughly presented in [7], all of which are implemented using modern model-based techniques. In the interest of space we do not show the formal anchoring process here, but rather describe the semantics of a specification informally. Our intention is to show a non-trivial CtC tool along with example systems, that, in the interest of space, are admittedly simple. In fact, we will use examples from the digital hardware realm because they are simple and canonical, but embedded software is designed using the same methodology. See [4], [11] for other examples of embedded software designed in the style of concurrent dataflow graphs.

2 SMOLES^2 Without Hierarchy

In order to introduce the SMOLES^2 approach, we begin by describing a subset of the total SMOLES^2 language. Specifically, we will temporarily ignore the instantiation of one SMOLES^2 specification inside of another SMOLES^2 specification, which is commonly called hierarchical composition. We consider hierarchical composition in Section 3.

A SMOLES^2 specification is divided into three interacting pieces called aspects. Each aspect provides a set of *design concepts* in an attempt to partition the system dynamics into manageable interacting pieces. A specification must

be constructed in a sequence where the first aspect is described, and then the second aspect is described, and so on. The aspects and the sequence of their construction correspond to typical design increments. The first aspect, called the *Dataflow Aspect*, specifies which computational objects are in the system (e.g. an ALU), and how these objects communicate. Constructing this aspect is analagous to constructing a datapath or dataflow graph. The second aspect, called the *Mode Aspect*, manipulates the datapath by disabling and enabling computational objects. This aspect is analagous to control logic, although it is not specified as an FSM. The third aspect, called the *Machine Aspect*, enables and disables parts of the control logic. This is analogous to a protocol specification and as such, encodes constraints on how the environment can interact with the system. After all three aspects are constructed, they may be modified in any order. These aspects of the specification incrementally "ramp up" the amount of non-linearity and time-varying behavior in a controlled manner, so that complex embedded systems can be specified.

While these aspects are common pieces of a digital system, our SMOLES^2 aspects are formally equivalent to SR systems. Thus, the designer constructs a system specification using a typical frame-of-mind (datapath,control logic, etc...), but is constrained to only produce SR systems, and in turn can apply well-studied analysis techniques. The SMOLES^2 specification tool was built on top of the metaprogrammable, model-based tool GME, so each aspect is defined by a UML-based *metamodel* written in the notation of *MetaGME* [6], and we will describe the language in terms of this metamodel. It is important to note that a metamodel is not just an abstract description of a modeling language, but can be used to generate a modeling environment that enforces the semantics of the metamodel diagram. By using a model-based metaprogrammable tool like GME we can rapidly construct a tool that supports our specification language, and we can utilize modern model-based concepts like aspects, generative actions, constraints, and type-instantiation that are core concepts in GME.

2.1 The Dataflow Aspect

The dataflow aspect, used to specify the simpliest layer of behavior, is a dataflow graph with three classes of dataflow objects: There are inputs that read data into the system, outputs that write data to the environment, and internal operators that perform computation. Internal operators have an interface that describes how many inputs are required by the operator and how many outputs are produced. An operator cannot produce output until it has data on every input, at which time it consumes all of its data, and produces data on every output. Internal operators are assumed to be stateless, though their behavior is not modeled in SMOLES^2. Rather, each operator has a place to store implementation parameters so that the operator can be correlated to its functionality. Figure 1 shows the part of the SMOLES^2 metamodel that defines operators. The *Operator* class has a number of attributes for defining visualization information and implementation information. Importantly, the attribute *UserDefinedClass* contains the name of the (external) class that implements an operator, and the attribute *ImplementationParam* contains a parameter that can be passed to an instance of the user-defined class.

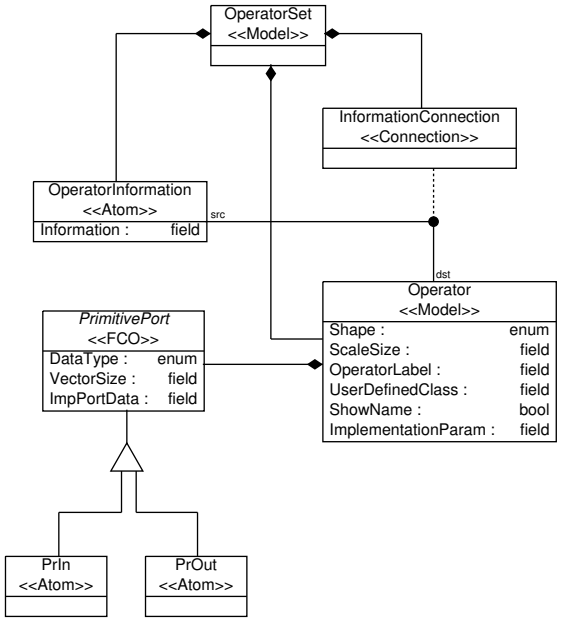

Figure 1. Operator definitions in SMOLES^2 metamodel, with example in the inset.

Each operator must have an interface that defines the number of inputs it requires, the number of outputs it produces, and the types of each I/O. This is defined in the metamodel by a containment relation between the *PrimitivePort* class and the *Operator class*. The *PrimitivePort* class characterizes the data type and size (in the case of a data vector) of an I/O, and provides an attribute *ImpPortData* so that an I/O can be matched with an I/O in the user-defined class. An operator does not contain instances of *PrimitivePort* directly, but contains one or more of the classes that inherit from *PrimitivePort*. These classes are *PrIn* that represents an input, and *PrOut* that represents an output. For the sake of organization, several operators are contained in an instance of the *OperatorSet* class, and operators may have documentation attached to them through an association with an instance of an *OperatorInformation* class.

Given a set of dataflow operators (instances of *Operator* class), these dataflow operators can also be instanti-

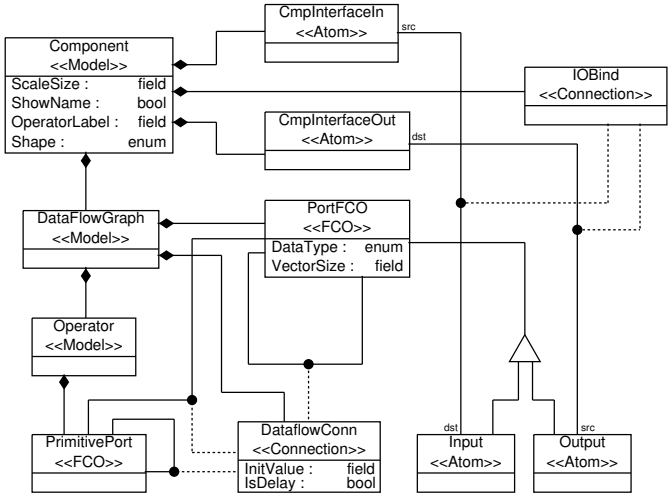

Figure 2. Dataflow graph and component definitions in SMOLES^2 metamodel.

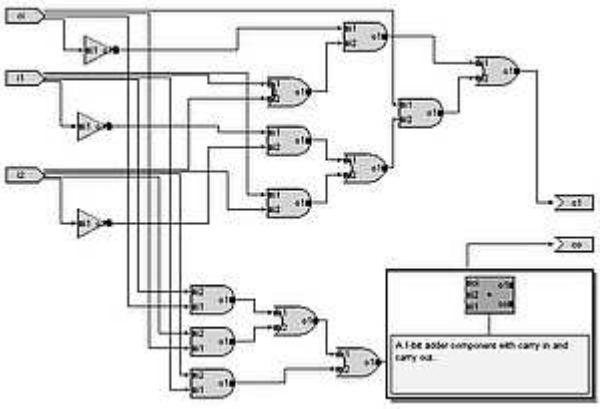

Figure 3. Example dataflow graph that implements a one-bit adder.

ated through type-instantiation. Type-instantiation results in another instance of an *Operator* class, but with the exact same internal structure, so if the AND gate were type-instantiated, it would produce a copy of an AND gate with technical restrictions on how that copy can be modified. (See [5], [10] for more details on type-instantiation.) We use type-instantiation to build dataflow graphs from operators. Figure 2 shows how graphs can be constructed. The class *DataFlowGraph* acts as a container for instances of operators, and for connections between operator ports. As mentioned earlier, a graph also has inputs that read data from the environment and outputs that write data to the environment. The classes *Input* and *Output* represent graph I/O, and are subclasses of *PortFCO*, which endows the I/O classes with data type and size attributes. Graph I/O objects can also be connected together, and can be connected to ports of operators. A connection can be *delayed* meaning that the connection stores the most recent value written to it, but outputs the previous value written to it. A delay connection must have an initial value. In the metamodel the *IsDelay* and *InitValue* attributes of the *DataFlowConn* class store this information.

The *Component* class is the top-level wrapper for a specification. It contains all three aspects of the specification, and so a component contains exactly one instance of *DataFlowGraph*. A component also exposes an interface, and this interfaces matches the interface on the dataflow graph. For each *Input* (*Output*) in the graph there is an instance of *CmpInterfaceIn* (*CmpInterfaceOut*) that is bound to that input (output). With these definitions we can rapidly construct a meaningful dataflow graph. For example, Figure 3 shows a one-bit adder with carry-in and carry-out signals. The inset shows the component when viewed from the outside. Notice how the graph inputs and outputs have been exposed at the component level. A component can also have documentation.

2.2 The Mode Aspect

The Mode Aspect allows one to specify non-linear behaviors that cannot be specified in the Dataflow Aspect. In terms of more familiar digital design, the Mode Aspect corresponds to control logic that is applied to the datapath. This control logic is not defined as an FSM, but rather by a set of modes where each mode can reconfigure the dataflow graph by disabling computational objects. More precisely, a mode is a directed graph that topologically resembles the dataflow graph, but its vertices describe whether or not the corresponding object in the dataflow graph should be disabled. This works by constructing a mode from *references* to dataflow objects. A reference is a modeling construct that allows an object in one model to refer (like a pointer) to another object in another model. A reference may also have its own attributes. In our specification tool, a mode is a graph with vertices that are references, such that each reference is annotated with an attribute indicating whether or not the referred dataflow object should be disabled. Figure 4 shows how this is described in the metamodel. For each class that can be contained in a dataflow graph, there is corresponding class that can be contained by a mode. For example an instance of a *ModeOperator* class refers to an instance of an *Operator*, as indicated by the arrow from the *ModeOperator* class to the *Operator* class. The *ModeOperator* class has an enumeration attribute *ShouldUseOp* which can take on three possible values. If $ShouldUseOp = Yes$, then the referenced operator will not be disabled from the

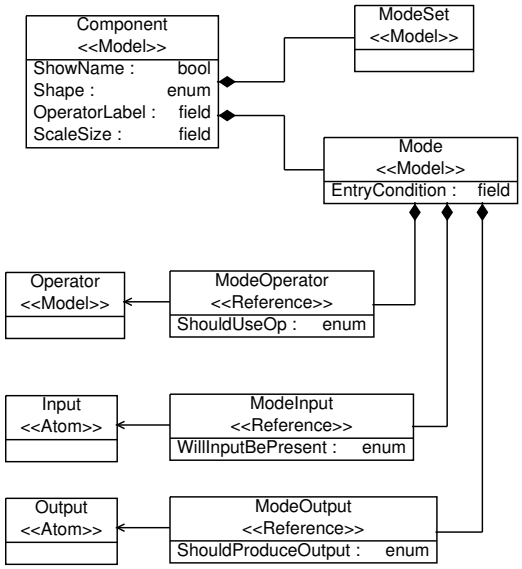

Figure 4. Mode definitions in SMOLES^2 meta-model.

graph. If $ShouldUseOp = No$, then the reference operator will be disabled in the graph. If $ShouldUseOp = Constant\ Operation$, then the dataflow operator is considered to be a constant value, which is disabled based on the disabled signals of its neighbors. Changing the enabled/disabled attribute of a *ModeOperator* changes its color: Black indicates a disabled vertex, gray indicates a constant vertex, and white indicates an enabled vertex. Figure 5 shows two modes for the 1-bit adder example. (Note that the edges have been hidden in Figure 5.) The left mode of Figure 5 leaves all of the objects in the graph enabled, while the right mode disables all of the objects in the graph.

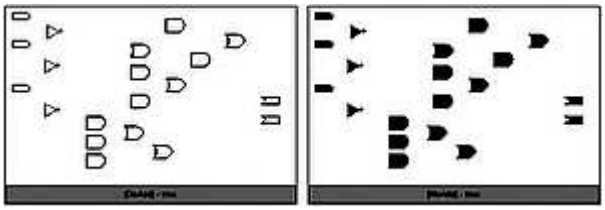

Figure 5. Two example modes for the 1-bit adder.

The mode set characterizes the ways the datapath can be configured, or equivalently, the set of control responses that a controller can make. Of course, a mode set is meaningless if there is no way pick which mode to apply. Therefore, a mode also describes when it should be applied. The mode encodes an *entry condition* in two ways. First, the enabled/disabled information on *ModeInput* instances (that refer to graph inputs) encode conditions on the availability of data. If a graph input i is enabled in mode m, then mode m is a valid control response only if the environment provides data on input i. In synchronous languages this a test on the *presence* of data. Conversely, if a graph input i is disabled in mode m, then mode m can only be applied if the environment does not provide any data for input i. This is a test on the *absence* of data. The mode on the left can only be applied if the environment provides data on all of the adder inputs ($i1, i2, ci$) and the mode on the right can only be applied if the environment provides no data for any of the inputs. An interaction with a component is valid only if some mode can be applied, so providing the 1-bit adder with only an $i1$ signal would result in a run-time error. This is a reasonable constraint, considering that the digital logic in Figure 3 would behave irradically if it were not provided all (or none) of its inputs.

Often times we want a control response to be applied if the data is present and carries a particular value. For example, we may want a division component to only perform division if the divisor is nonzero, so we could write the entry condition $divisor$!= 0. Such data-dependent entry conditions are specified in the *EntryCondition* attribute of a mode. SMOLES^2 supports the condition language shown in Figure 6 (assuming that signals are bit vectors). The

```
expr   ← '(' expr ')' | var
       | bitvec | bool
       | expr numop expr
       | expr binop expr
       | expr bitop expr
       | expr '[' number ']'
       | expr '[' number '-' number ']'
numop  ← '='  |  '>' |  '<' |
       '!=' | '>=' | '<='
bitop  ← '|=|' | '|!|'
binop  ← '&' | '|'
var    ← [a-z,A-Z,_ ][a-z,A-Z,0-9,_ ]*
number ← [0-9]+
bitvec ← [0,1]+
bool   ← 'true' | 'false'
```

Figure 6. BNF-grammar for the condition language supported by SMOLES^2.

numop operators perform numerical comparisons, while the *bitop* operators compare bit vectors. Slicing operators extract sub vectors from a bit vector, so var[0-4] extracts the subvector from positions zero to four, inclusive.

As a final note, notice that our modes expand the set of behaviors that a component can exhibit. In other tools modes serve as a convenient way to group together behaviors, while in our tool we must have modes in order describe true SR systems and not just homogenous SDF systems. Of course we must be careful not to make modes so expressive that unanalyzable systems could be specified. In order to prevent this we only allow modes to enable/disable vertices in the SDF graph.

2.3 The Machine Aspect

We used the Mode Aspect as means to specify non-linear dynamics. Similarly, we use the Machine Aspect as a means to specify time-varying dynamics. Continuing with the digital design analogy, control logic is naturally specified by modes that enable/disable dataflow operators, while state-dependent interaction protocols are naturally described by state machines that enable/disable modes in the mode set. A Machine Aspect is a state machine where each state enumerates a subset of modes from the mode set. When the system enters a state, it disables all modes except those enumerated by the state. At this point either the environment interacts with the system so that a mode from the subset can be applied, or a run-time error occurs. If a mode is applied, then the state machine takes a transition to another state. Transitions only have *mode application events* as guards. A transition $s_i \xrightarrow{e_{m_k}} s_j$ means that the machine goes from state s_i to s_j if the event where mode m_k was applied occured while the system was in state s_i. Intuitively, a Machine Aspect encodes a set of control trajectories, and the environment must remain on one of these trajectories, otherwise a run-time error is produced. Figure 7 shows how a machine is represented in the SMOLES^2 metamodel. A state, which is an instance of a class *State*, contains a collection of references to modes. State transitions, which are instances of *StateTransition*, begin on mode reference and end on state. The interpretation is that the state that contains the reference is the source state (s_i), the mode pointed to by the reference is the guard (m_k), and the destination state is the destination for the transition (s_j). A machine must contain exactly one start state, which is an instance of the class *StartState*, that has a guardless transition to some state. The start state does not contain any modes, and its transition is taken immediately.

2.4 Well-formed Models

With this background we now describe how our CtC tool SMOLES^2 detects and corrects specifications. We use two model-based approaches to detect and correct models, and these can be broadly classified as generative and constraint-based. The generative approach detects changes in one aspect of the specification and then *generates* changes in other aspects to make the overall specification consistent. The constraint-based approach checks partial specifications for problems, localizes those problems, and reports them to the user. The constraint checks are *conservative approximations* in the sense that a constraint violation implies that verification will fail, but if a constraint is not violated then verification may or may not fail. Constraint checks are polynomial time algorithms that operate directly on the model syntax, i.e. they do not check the model with respect to a different representation. This means that they can identify the malformed part of the specification but they only approximate the true verification algorithms, which are NP-hard.

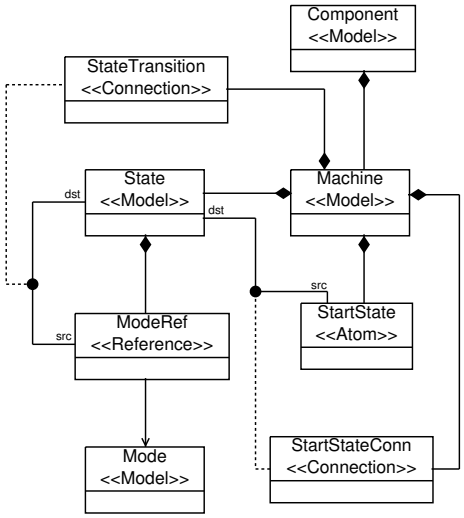

Figure 7. Machine definitions in SMOLES^2 metamodel.

Generative actions work to maintain a consistent specification across aspects. The generative actions are simple, but absolutely essential to the scalability of the SMOLES^2 specification language. For example, given a dataflow graph with n inputs, we may construct 2^n unique modes without using data-dependent entry conditions (and many more with data-dependent entry conditions). If the dataflow graph is changed, it may mean changing up to 2^n modes so that the topology of each mode matches that of the dataflow graph, and so that the references in the modes point to valid objects in the dataflow graph. The engineer cannot be expected to maintain these consistency requirements, so SMOLES^2 includes a generative engine that watches changes to the dataflow graph and updates all modes so that they become consistent with the dataflow graph. This means that if an object is added to the dataflow graph, then a new reference is created in every mode. If an object is deleted, then the ref-

erences are deleted from every mode, and if a connection is created/deleted, then a connection is created/delete between references in every mode. Generative actions are also used to generate component interfaces and to remove references to deleted modes from states in the Machine Aspect.

Checking the Dataflow Aspect and the Machine Aspect is fairly simple. First, a type check on edges in the dataflow graph ensures that connections between ports respect signal types and widths. As a necessary condition for bounded memory, we also check that every primitive operator has all of its inputs fed by some other operator or input, i.e. we check that there are no dangling inputs. As a necessary condition for determinism, we check that there is a start state, and that there is exactly one transition for every mode reference.

Checking the Mode Aspect is a more complicated process. Disabling an object from a dataflow graph can have the negative effect of causing a dangling input on another operator. We must check that disabled operators do not induce dangling inputs in the dataflow graph. The lower left-hand mode in Figure 8 induces such dangling input on the bottom AND gate. Another problem arises when

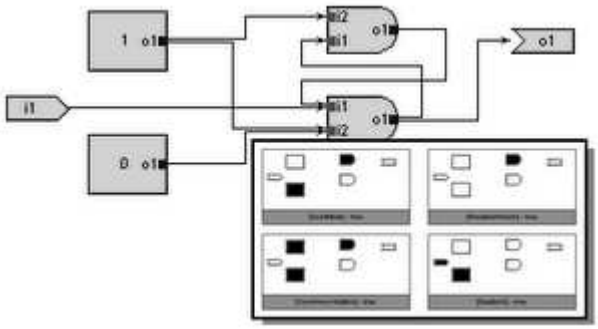

Figure 8. Dataflow graph exhibiting potential problems.

dataflow operators are connected in a cycle (without any delay edges). If all operators in the cycle are enabled in a particular mode, then deadlock results. In order to prevent deadlock, every mode must disable one or more objects in the cycle. Checking this property is equivalent to the *causality analysis* of the synchronous languages, if no hierarchy is in the SMOLES^2 model. When hierarchy is added, this check becomes a conservative approximation of causality analysis. The lower right-hand mode in Figure 8 permits the deadlocked cycle that includes the two AND gates. A third problem arises if multiple operators write data to the same input port of another operator. Since the system is supposed to be deterministic, every mode must disable one or more of the operators that drive the same input port. The upper right-hand modes allows the values 0 and 1 to be simultaneously written to the 2^{nd} input of the bottom AND gate, and this violates the determinism requirement. The upper left-hand mode eliminates all of these problems.

We check for these problems by assigning a *control type* to each vertex in a mode, and then by performing type inference on the control types. Mode vertices that reference graph inputs can be of type *absent (*$\mathbf{A}$*), present (*$\mathbf{P}$*), constant (*$\mathbf{C}$*),* or *don't care (*$\mathbf{D}$*).* (A *don't care* input allows a mode to be entered regardless of whether data is present or absent on that input.) All other vertices can only be of type *absent, present,* or *constant.* The inference works by first assigning a control type to all non-deterministic merges. Let the sequence $t_1, t_2, \ldots, t_n$ denote the control types of all vertices participating in the merge, then the type of the merge is $t_{merge}(n)$. This is calculated according the following recursion:

$$\begin{array}{c} t_{merge}(1) = (t_1)_1 \\ t_{merge}(i>1) = (t_i)_1 \ \vee_{merge} \ t_{merge}(i-1)_2 \end{array} \quad (1)$$

where the operator $\vee_{merge}$ is the join of the *merge lattice* shown in Figure 9. For technical reasons involving the

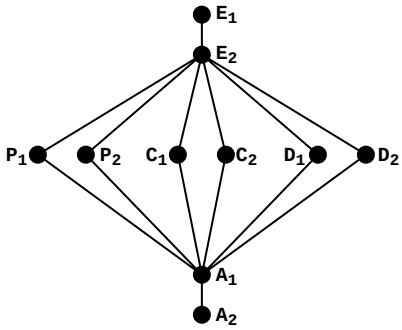

Figure 9. Merge lattice for type-inference of a merge.

symmetry of the join operation, each control type is split into two different elements of the lattice, and the recursion "casts" between these elements. For example the *absent* type can be cast to $\mathbf{A}_1$ or $\mathbf{A}_2$, and we denote this casting with the operators $(\bullet)_1$ and $(\bullet)_2$. The elements $\mathbf{D}_1$ and $\mathbf{D}_2$ are the *don't care* types, and the elements $\mathbf{E}_1$ and $\mathbf{E}_2$ are the *error* types. If the type of a merge is the error type, then the merge is illegal, and this happens if the control types imply that more than one value will be written to the port. After every non-deterministic merge has been assigned a type, we can aggregate all of the types on the input ports of each operator to form the *apply* type t_{apply}. Calculating t_{apply} uses the same recursion as Equation 1, except the join is over a different lattice called the *apply semi-lattice* (Figure 10).

This semi-lattice ensures that no dangling inputs will occur. Notice how constant data can play the role of a present or absent signal, depending on context. Given an operator with input ports $p_1, p_2, \ldots, p_m$, let the sequence $t_1, t_2, \ldots, t_m$ be the control types on each port. If the port p_j has a merge, then the port type t_j is the type of the merge, otherwise t_j is the type of the single vertex writing to port p_j. Using this

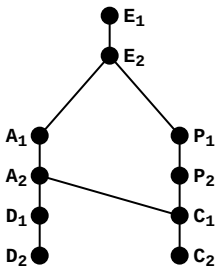

Figure 10. Apply semi-lattice for type-inference of input ports.

sequence and the apply semi-lattice, calculate $t_{apply}(m)$ to find the apply type. If the process of determining the merge and apply types yields an error type, then that error is localized to the vertex that first caused the type to switch to an error type. After all merge and apply types are calculated (and assuming no errors are found), we check that the type t^v of a vertex v is compatible with the type being applied to that vertex (t^v_{apply}). The types are compatible if the following holds:

$$\begin{gathered}\Big((t^v = \mathbf{P}) \Rightarrow (t^v_{apply} = \mathbf{P}) \vee (t^v_{apply} = \mathbf{C})\Big) \\ \wedge \\ \Big((t^v = \mathbf{C}) \Rightarrow (t^v_{apply} = C)\Big)\end{gathered} \tag{2}$$

If Equation 2 does not hold, then the error is localized to the vertex v. All of these checks are polynomial time and occur in response to modeling events or can be invoked by the engineer. Furthermore, the checks operate on the specification and localize errors to the offending parts of the specification.

2.5 Examples

The first example is a 1-bit memory with read (R), write (W), 1-bit input (I), and 1-bit output (O) signals (Figure 11). Though this example is simple, it uses many constructs including cycles, merges, and *don't care* inputs. The small rectangular objects are identity operators (zero-delay buffers) that pass data through without modification. Two of these identity elements are connected in a cycle with a delay edge (the dotted line) in order to hold a bit of data. A write occurs when R is absent, and data is present on the W and I signals. The Write mode (upper left-hand mode) disables the third identity function, which breaks the feedback cycle and allows the input data to be written onto the delay edge. The Read mode (upper right-hand mode) is applied whenever R is present. The dark gray coloring on the

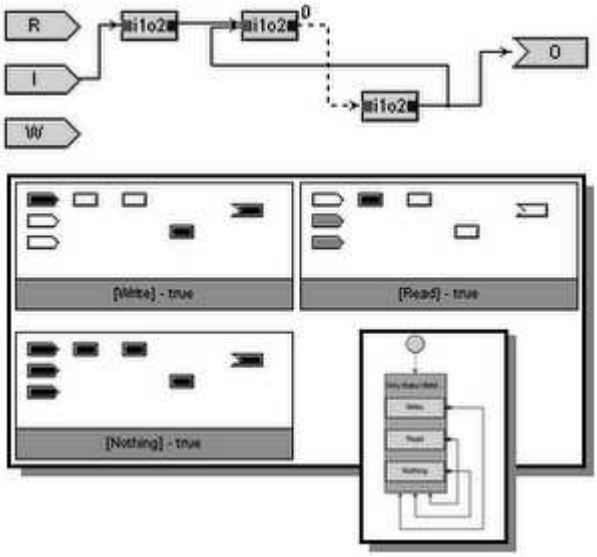

Figure 11. 1-bit memory example.

W and I signals indicates *don't care* conditions. The Read mode disables the first identity function, so that if data is present on the I signal it cannot disturb the bit stored in the feedback cycle. The small inset shows the Machine Aspect, which only has one state with all three modes permitted by the state. Each mode returns the system back to the same state.

In the second example we consider a system with a more interesting Machine Aspect. Figure 12 shows a 2x clock multiplier (or oversampler). The input to the system is a clock signal CLK and the output is another clock signal $FAST$ with a rate twice that of CLK. The dataflow graph writes the constant value 1 to the output $FAST$. The machine starts in the state $Tick$ which restricts the system to the upper left-hand and lower left-hand modes ($TICK$ and $NOTHING$). While the environment provides no data for CLK, the system stays in the $Tick$ state. When a clock tick occurs, the $FAST$ signal emits a value, and the system goes to the $OverSamp$le state. In the $OverSamp$ state only the upper right-hand mode can be applied ($OVERSAMP$), which requires the environment to provide an event without a clock tick and responds with a clock tick on $FAST$. After emitting $FAST$, the system returns to the $Tick$ state and the process repeats. The result is that for every tick of CLK there are two ticks of $FAST$.

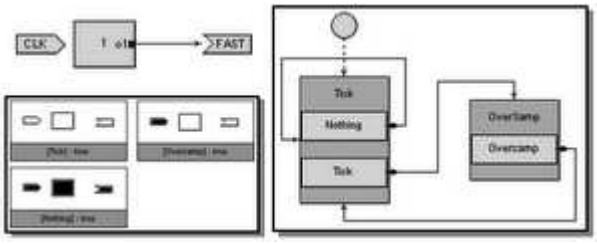

Figure 12. 2x clock multiplier example, i.e. oversampler

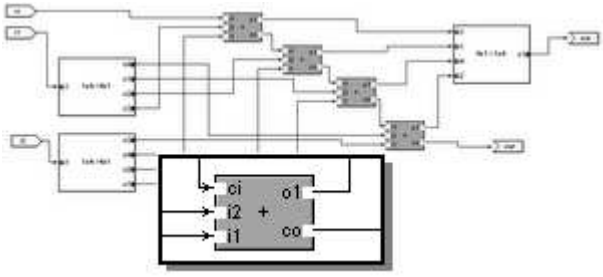

Figure 13. ADD mode for 4-bit adder.

3 SMOLES^2 with Hierarchy

Hierarchical composition is essential for the scalability of our (and most other) specification techniques. Hierarchical composition allows one to use a previously defined SMOLES^2 specification (component) in a dataflow graph. Replacing an operator with a component has a profound effect on the methods that we previously described. In general components are not *monochronous*, i.e. a component can respond to an event that does not provide data to every input on the component's interface, and a component need not provide output on every output port. Contrast this with the primitive operators that can only fire when every input has data, and when fired, produce data on every output. Components also have state in the form of delay edges and the current state of the Machine Aspect, while operators are stateless. Nonetheless, we can still assign a reasonable semantics to this composition: Operators fire whenever they have data on all of their inputs, and components fire when every input has data or when all writers of that input have been disabled. In another words, a component can fire once it is known which of its inputs are present and which are absent. This definition eliminates all cycles that consist entirely of components, and this constraint can be easily checked.

In the previous section we used control types to statically determine which operators would be scheduled, which would be disabled, and if the schedule was legal. Similarly, we can extend the typing mechanism on mode vertices to figure out which inputs on a component's interface will have data and which will be absent of data. Once we know this information, we check if that component contains a mode that would accept that particular combination of absent/present data. If we find such a mode that accepts the input conditions, then we can determine the presence and absence of the output ports for that component according to the matched mode. This information is then placed on the output ports of the instantiated component, and the process continues. An example of this is shown in the mode in Figure 13. This mode is a control response for a 4-bit adder built by instantiating four 1-bit adders as shown in Figures 3 and 5. The 4-bit adder rips two 4-bit signals into eight 1-bit signals, and then feeds a pair of bits into each 1-bit adder (gray boxes). The carry-outs are chained through the 1-bit adders, and the outputs from the four 1-bit adders are repacked into one 4-bit signal. The inset shows a zoomed picture of a 1-bit adder component as it appears in the mode. Notice that the inputs and outputs of the component are colored white. This coloring indicates that in this mode the 1-bit adder will have data present on all of its inputs, and will respond by producing data on all of its outputs. In the mode where no data is presented to the 4-bit adder, all of the inputs and outputs on all 1-bit adders would be black.

In general, it cannot be statically determined which mode, if any, a component will apply in response to a particular set of present/absent inputs. There are several reasons for this. First, as in the case of the clock multiplier (Figure 12), the particular response may be dependent on the state in the Machine Aspect. Second, if a component has modes with data-dependent entry conditions (e.g. $i1 > 0$) then we cannot predict which modes can be applied because we do not know the values of the data. We handle this by applying conservative approximations to predict whether or not a mode exists that will accept the inputs. The approximation we apply is to consider the entire set of modes (i.e. ignore the Machine Aspect) with all data-dependent entry conditions set to true. If no mode from this relaxation accepts the inputs, then no mode exists under any state/data pair that will accept the inputs. If the component has no data-dependent entry conditions, and a trivial Machine Aspect, then this approximation becomes exact.

This approximated response must be handled carefully, because we may identify several modes that can be applied, and these modes may contradict which objects in the graph are enabled/disabled. For example, if we know that the clock multiplier does not receive a *CLK*, then the *OVERSAMP* mode asserts the output $FAST$ to be present while the *NOTHING* mode asserts the output to be absent. Since we do not have enough information to choose the correct

mode, we must assume that either situation is possible. We formally encode this missing information by placing a new control type onto outputs with contradictory disable/enable information. This *don't know* control type (abbreviated **K**) indicates that we "don't know" if the particular output will have data. We must modify the type lattices so that type inference handles the *don't know* type. (We do not list the modified lattices.) Figure 14 shows a mode with an instantiated clock multiplier that has an absent *CLK*. Notice that the output port of the clock multiplier is colored half white and half black. This is the visualization of the *don't know* type, which is propagated to the output of the parent component.

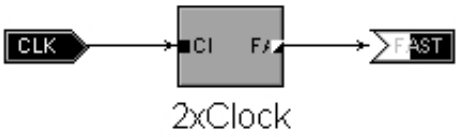

Figure 14. Example of a *don't know* type.

The generic algorithm for checking a mode in hierarchical SMOLES^2 combines all of these techniques. First, we check the mode for any cycles of only components, and for any present/constant cycles of operators. Next, we calculate t_{merge} for every port and t_{apply} for every operator for which we have complete type information. If we do not have complete type information, then this means we must approximate the modes of various components. In order to perform this approximation in a correct order, we evaluate and propagate component output types according to a topological sort of the subgraph that only contains components.

4 Conclusions and Future Work

We presented a detailed CtC tool for specifying a particular class of embedded systems, and we provided polynomial time conservative approximations for checking the validity of specifications. These approximations are based on well-known graph algorithms (e.g. cycle checking, topological sort) and type inference techniques (e.g. type lattices). Additionally, we showed how generative actions can be used to make interacting aspects consistent without user intervention.

Our current and future work is to build code generators that convert a SMOLES^2 specification into input for a verification or CbC tool. We are approaching this from several fronts. One front is to write a verification tool that performs verification in a layer-wise fashion that matches the sequence of aspects in a SMOLES^2 specification. The other natural front is to map our specifications to existing synchronous languages like Esterel [3] and Signal [12], for which many tools already exist. Once connected to these tools, this CtC tool will be merged with existing IuV and CbC tools.

References

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

[2] R. Bergamaschi and J. Cohn. The a to z of socs. *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, pages 790–798, 2002.

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

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

[5] A. L. J. G. G. Karsai, M. Maroti and J. Sztipanovits. Type hierarchies and composition in modeling and meta-modeling languages. *IEEE Transactions on Control System Technology*, 2003.

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

[7] E. K. Jackson and J. Sztipanovits. Using separation of concerns for embedded systems design. In *Proceedings of the Fifth ACM International Conference on Embedded Software (EMSOFT'05)*, September 2005.

[8] S. N. M. E. K. Chen, J. Sztipanovits and S. Abdelwahed. Toward a semantic anchoring infrastructure for domain-specific modeling languages. In *Proceedings of the Fifth ACM International Conference on Embedded Software (EMSOFT'05)*, September 2005.

[9] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[10] E. A. Lee and S. Neuendorffer. Classes and subclasses in actor-oriented design. *Conference on Formal Methods and Models for Codesign (MEMOCODE)*, June 2004.

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

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

[13] A. L. S.-V. R. Passerone, J. R. Burch. Conservative approximations for heterogeneous design. In *Proceedings of the Fourth ACM International Conference on Embedded Software (EMSOFT'04)*, September 2004.

[14] S. R. Schach. *Object-oriented and classical software engineering*. McGraw-Hill, fifth edition edition, 2002.

[15] B. H. T. A. Henzinger and C. M. Kirsch. Giotto: A time-triggered language for embedded programming. In *Proceedings of the First International Workshop on Embedded Software (EMSOFT)*, pages 166–184. Lecture Notes in Computer Science 2211, Springer-Verlag, 2001.