Heterogeneous Modeling and Simulation of Embedded Systems in El Greco

Joseph Buck and Radha Vaidyanathan
Synopsys, Inc.
{jbuck,radha}@synopsys.com

Abstract

This paper describes the functional specification and verification portions of El Greco, a system for high-level, heterogeneous functional specification, efficient compiled simulation, and software and hardware implementation. Specifications in the form of dataflow graphs, hierarchical finite state machines, or a mixture, are supported. These specifications can be arbitrarily nested, as in Ptolemy [1]. When dataflow graphs are placed in a control context, the graph execution is fully controllable; its execution can be restarted or suspended and parameters can be changed. We describe system modeling and simulation generation in El Greco and compare to other approaches.

1. Introduction

Designers of complex embedded systems are under pressure to deliver more and more functionality in less and less time. Designing at a higher level of abstraction has long been accepted as a means of achieving greater productivity. However, it is generally accepted that the needed abstractions are domain-specific, and that these abstractions need to be efficiently coupled, inspiring work like that of the Ptolemy project [1]. El Greco's focus is on the design of system-level tools for the current generation of digital communication and multi-media systems, and we find that control and dataflow operations are commonly mixed at multiple levels of hierarchy, making it difficult to decompose systems into control-dominated and dataflow-dominated components each to be designed by an appropriate tool.

It is generally accepted that functional verification dominates the time required to design, implement, and test an embedded system. It is not sufficient to simply be able to model heterogenously; the functional simulation of such models must be extremely efficient and implementation from high levels of abstraction must avoid the synthesis of redundant hardware of software. We believe the simulation-based approach to heterogeneity used by Ptolemy will not be sufficient to achieve this, unless we can optimize across domain boundaries.

El Greco's core concepts are:

- Provide a range of modeling styles so that the right abstraction can be used and details added as design progresses by performing data and control refinement.

- Provide the ability to nest models at will, at any level of hierarchy (inspired, of course, by Ptolemy).

- Base the modeling concepts on principles with rigorous semantics: dataflow and synchronous-reactive languages such as Esterel [2].

- Specify conditions, actions, and other behavior in a C++ subset, which is parsed and understood by the tool.

- Use detailed analysis and compilation to achieve very high simulation speed. Dead code is removed and transformations take place across control/dataflow boundaries.

- Use internal representations that are suitable for hardware, embedded software, or compiled simulation generation from the same models. In particular, actions that are specified in C++ can be emitted to back end tools in VHDL or Verilog.

- Provide a state-of-the-art user interface that simplifies the capture of complex heterogeneous systems, either in a top-down or bottom-up manner (not shown in this paper).

El Greco is designed to provide a path to implementation as embedded software, synthesizable hardware or both. This paper describes only modeling and functional simulation, though implementation issues figure strongly into the design.

2. Relationship to previous work

This work is strongly influenced by a number of systems that came before it; particularly strong influences include Ptolemy [1] for heterogeneous design, and Esterel [2] for control semantics. While this work was performed independently of Ptolemy's *-charts [3], there are many resemblances; we make detailed comparisons in section 3.1. The semantics of hierarchical state machines resembles that of Andre's SyncCharts [4], which is in turn a variant of Harel's StateCharts [5].

2.1 Combining dataflow and control

There are many hybrid modeling approaches in the literature that, in some sense, embed FSM-style specification with dataflow-like data communication. We see two types of approaches: that of Ptolemy [3], which allows arbitrary nesting, and that of almost all other tools, which restrict the form of the heterogeneity. We follow a Ptolemy-like approach in our work; differences will be described in a subsequent section.

SDL [6] and Polis [7] specify local behavior in the form of extended finite state machines and then use some form of asynchronous inter-process communication between these local behaviors. Alternatively, a supervisory controller controls the execution states of a connected graph of modules, where these modules communicate in a dataflow style (e.g. [8]). In [9], a dataflow system is enhanced by adding event-style communication

between finite state machines and dataflow actors; this introduces nondeterminism which must be controlled by a knowledgeable user who specifies scheduling constraints.

These techniques differ in details: the communication channels might be a one-place buffer or a FIFO queue, for example. However, the heterogeneity is constrained; the nesting of models of computation always follows the same form (e.g. primitive FSM-like blocks at the innermost level, surrounded by inter-process communication between such primitives).

The "bottom level", where a synchronous-reactive behavior is typically specified, may itself be specified in a heterogeneous manner, using a language like Esterel, or regular expressions [10] to structure the control and a software language (e.g. C) or a hardware implementation language to specify atomic actions to be executed by the controller. ECL [11] permits the use of both types of constructs (reactive and computational) to be specified in one language, as does SystemC (formerly called Scenic [12]).

2.2 Control specification

Our control modeling techniques have their roots in those of Harel's Statecharts [5]. There have been problems with the semantics of Statecharts, resulting in a large number of Statecharts variants [13]. Our control models closely resemble those of Andre's SyncCharts [4]. The detailed formal semantics of control models are defined in terms of a translation to an Esterel equivalent. We have chosen strictly synchronous semantics to enable a complete analysis at compile time, resulting in more efficient compiled simulation and hardware implementation.

2.3 Dataflow specification

El Greco uses cyclo-static dataflow (CSDF), which was pioneered by the Grape-II project at the Catholic University of Leuven [14]. Dynamic dataflow is also supported. The use of CSDF permits finer-grain control of multirate dataflow graph execution than is possible with synchronous dataflow.

3. Modeling in El Greco

In El Greco, the user composes designs in either a top-down or a bottom-up fashion, by instantiating models. Models can be primitive or hierarchical.

A primitive dataflow model, or prim_model, may read or write data from or to ports, and can have internal state. Dynamic dataflow and static dataflow models are written in exactly the same way; analysis determines whether a prim_model is dynamic or not.

There are four types of hierarchical models:

Dataflow graphs. These are graphs whose nodes are instances of other models; the ports of the instances are connected by nets. A net has one driving port and can have any number of destination ports. The communication is FIFO data streams; the streams are duplicated if there are multiple outputs on a net.

Or-models. An or-model is a collection of mutually exclusive states with transitions between states. The states can be atomic or can be instances of other models. Atomic states may optionally have an inline action, which is a statement of C++. If the member states of an or-model are all atomic, it represents a flat FSM. The states are connected by transitions, with conditions and actions, written in C++.

And-models. An and-model represents a group of models that execute in parallel lock-step with broadcast synchronous-reactive communication between them. We use the constructive semantics of Esterel to determine fixpoints in the event of cycles [16] and reject cases that do not have unique fixpoints at compile time.

Gated models. A gated model has one or two children, plus a gating condition, which can depend on ports and parameters of the model. If the gating condition is true, the first child is run and the second (if any) is suspended. If the gating condition is false, the second child (if any) is run and the first is suspended.

We use the term *control model* to describe the three types of hierarchical models that are not dataflow models.

For the most part, the user of a model need not be concerned whether that model is implemented as a hierarchical dataflow graph, or-model, or primitive dataflow model (for example); the external interface looks the same in all three cases. In El Greco there is a clear separation of the interface of a model from its implementation. A graphical symbol for a model is automatically generated (which then can be customized).

In tools such as Ptolemy and COSSAP, model execution has three phases. In the reset phase, parameter values are obtained and checked and the model is initialized. In the main phase of execution, the model reads and writes data. In the final phase, a wrap-up occurs, for freeing resources, writing final results, etc. However, all instances are reset at the beginning of a simulation run, and all are "wrapped up" at the end.

When hierarchical control is added, it becomes natural to reset or wrap up instance execution at multiple points (for example, where a hierarchical state of an or-model is entered or exited). We support this, and go beyond it in that parameters of instances can be changed at the point of entry.

In El Greco, an instance of a model can be terminated by its parent in one of two ways, so called strong termination and weak termination. These concepts are borrowed from Esterel, where the term "abort" is used instead. If an instance is terminated with strong termination, it does not participate in the execution cycle where the strong termination event occurs; with weak termination it does. Gérard Berry has suggested an analogy to the Unix "kill -9" and "kill" events; for the latter there can be a cleanup handler; for the former no such handler is possible. When an instance is terminated its internal state, if any, is lost.

In addition, an instance can "voluntarily" exit. This may in turn cause the parent model to exit, or can be caught, like an exception, and trigger an action in the parent. Again, any internal state is lost.

Finally, an instance can be suspended. The effect is much like gating the clock of a hardware component: state is preserved, but the instance does not "see" input events until the suspension ends.

The interface of a El Greco model has:

Zero or more *ports*. Ports provide the only data communication path between models. In a dataflow context, ports are logically connected to FIFO queues (though, where possible, such queues are replaced by buffers of size one in the compiled simulation). In a control context, ports are bound to signals belonging to the parent model. Ports have a type and a direction (in, out, or inout). Models with inout ports cannot be used in a dataflow context, but can be used in a control context.

Zero or more *type parameters*. Rather than requiring one model to add integers and another to add bit vectors, type parameters permit models to be written once for all types.

Zero or more *parameters*. Parameters provide read-only data to a model and are an extension of the parameters seen in tools like Ptolemy, or generics in VHDL. However, to make control more powerful, we permit parameters to be changed under certain

circumstances by the control environment (which can be a control model in the simulation, or the top-level supervisor). Parameters may have several modes:

- *structural parameters* must be constant; their values are compiled into the simulation.

- *read-on-reset parameters* are sampled when the model they belong to is reset; their value can be changed by the control environment (e.g. by binding a read-on-entry parameter of a dataflow graph to a local signal in a control model). A typical example of such use is where the result of one dataflow computation (say, a signal detection routine) determines the parameters for a subsequent computation.

- *dynamic parameters* can be changed at any time; this feature is used to make simulations interactive and has an effect similar to the *volatile* keyword in C/C++.

3.1 Comparison to Ptolemy *-charts [3]

This work resembles [3], but there are important differences.

Ptolemy takes a simulation-oriented approach to heterogeneity: it has strong heterogeneous simulation capabilities but rather limited implementation or compiled simulation capabilities for models other than dataflow. We take a more implementation-oriented approach, meaning that models are generally speaking "white-box" and completely analyzed at simulation generation time (though we do use a Ptolemy-like approach to interface with hardware description language or instruction set simulators).

In Ptolemy, models intended for interpreted simulation are written in a completely different manner from models intended for software code generation (or compiled simulation) or hardware synthesis. In El Greco, all implementation models are simulatable.

As in Esterel, El Greco does fixpoint analysis at compile time to avoid the need for VHDL-style microsteps at simulation runtime.

El Greco can do static dataflow scheduling even in the presence of symbolic data rates (within limits); Ptolemy cannot do this.

El Greco is designed to permit changes to parameters of instances based on control model behavior.

Like *-charts, we permit the user to choose different models of concurrency (e.g. dataflow vs. synchronous-reactive). However, we believe that neither we nor they entirely succeed in making concurrency completely orthogonal to FSM sequencing, since in a hierarchical FSM a child instance is concurrent with its parent.

4. Cyclostatic analysis of prim_models

Primitive dataflow models are analyzed to determine whether they are statically schedulable. This is accomplished by attempting to divide the main action into phases and groups of phases. A phase is a segment of code that reads one value from some set (possibly empty) of input ports, optionally performs a computation, and then writes one value to some set (possibly empty) of output ports. We can also have phase groups, typically delimited by *for* statements. If we can deduce the number of times the loop is executed, and this number is a constant or depends only on parameters, we have identified a phase group.

An example of statically schedulable prim_model follows. First we have a phase group of Factor phases that reads the input; it is followed by a single phase that writes the output.

```
prim_model DownSample {
    type_param T = float;
    port in T InData;
    port out T OutData;
```

```
    param read_on_reset unsigned Factor;
    main_action {
        for (unsigned i = 1; i <= Factor; i++)
            read(InData);
        OutData = InData;
        write(OutData); }}
```

prim_models whose I/O pattern depends on data are not cyclo-static, and usually result in dynamic scheduling.

5. Dataflow graph scheduling

While there are some exceptions, control models, as a rule, look like unit-rate static primitive dataflow models to the dataflow scheduler and are treated as such. If the dataflow graph contains other dataflow graphs, we flatten them, to obtain one flat graph consisting of primitive models or control models.

El Greco's dataflow scheduler performs transformations on adjacent instances in the dataflow graph, in an attempt to collapse pairs of instances into clusters, which are then treated as instances themselves. There is insufficient space to describe the algorithm in detail here, but it is closely related to the clustering algorithm described in [15]. A qualitative description will have to suffice, as a full description would require a complete paper.

We alternate between a merge pass, in which adjacent instances with matching data rates are merged wherever this will not lead to deadlock, and a loop pass, where transformations are applied to instances to create more opportunities for subsequent merges. Possible transformations include "stalling", where empty phases that move no data are added, "sum-up", where a loop is added to combine all the phases of an instance into one phase, and "do-while", where a do-while loop is added that repeatedly executes an instance until a data value emerges or is consumed.

The merging and looping transformations are applied to the internal syntax trees that represent the actions of each model instance; dead code is eliminated from the merged clusters. In most cases the use of CSDF means that buffers of size one suffice for communication, so the generated simulation code typically passes values between instances in registers.

6. Mapping of Control constructs to Esterel

The semantics of El Greco control models closely resemble those of Esterel modules, although the user is not exposed to Esterel. Control models can have variables and signals, which are identical to those of Esterel, with the following difference: for valued signals, we do not use the presence bit, but only the value. As a result, we have two kinds of boolean signals: non-latching, corresponding to pure signals in Esterel, and latching, corresponding to valued boolean signals. We also permit the use of general expressions in contexts where Esterel requires Boolean combinations of pure signals; we hide the distinction between, for example, Esterel's *present* and *if* statements with code generation.

And-models correspond to the || (parallel) statement of Esterel: the child models execute in parallel lockstep, and the and-model exits when all of its children exit. Gated models correspond to the *suspend* construct of Esterel, combined with parallelism. In dataflow contexts, in most cases we can transform a gated model inline to an *if-then-else* construct.

Or-models are state transition graphs, resembling those of Statecharts, but much closer to those of SyncCharts [9]. They differ from those of SyncCharts chiefly in that dataflow graphs may be embedded inside. As in Statecharts, start transitions, possibly with conditions, select the initial state. Transitions are enabled by a
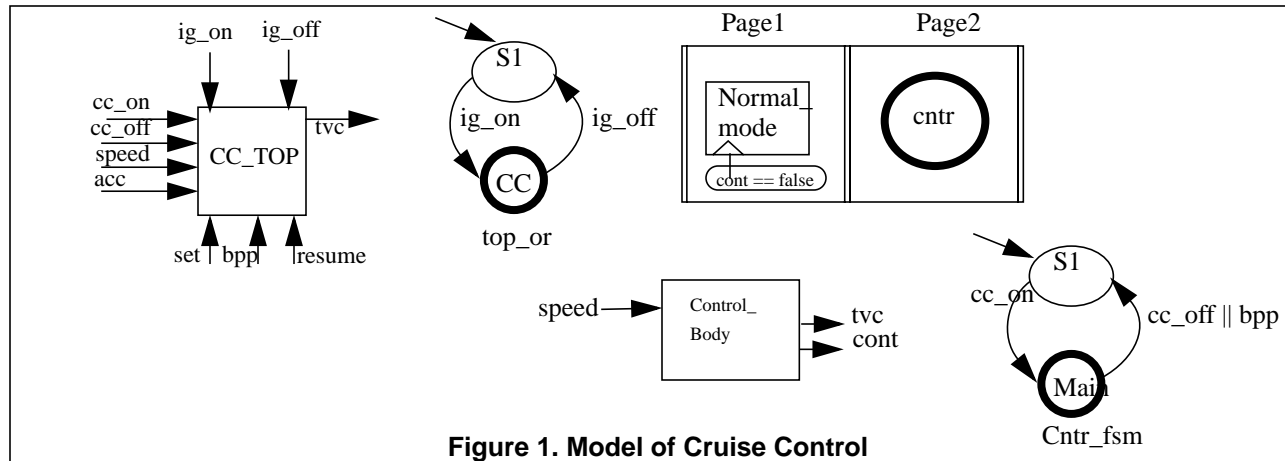
Figure 1. Model of Cruise Control

condition on the arc. There are three primary types of transitions: a strong termination transition, a weak termination transition, and an exit-handling transition. The first two types of transition terminate the execution of the currently active instance. Strong termination is equivalent to the *abort* statement of Esterel; weak termination is equivalent to the *weak abort* statement of Esterel (or, equivalently, a combination of *trap* and *exit*). Exit-handling transitions are enabled when the instance they are associated with exits and the optional condition on the transition is satisfied.

If more than one transition can be active at the same point, strong transitions take priority over transitions of other types; users can also specify priorities. The existence of two equal-priority enabled transitions is considered an error.

An or-model exits if an exit transition is taken. This is a transition with a source instance but no destination instance. The or-model also exits if the active instance exits, but there is no enabled exit-handling transition.

Because of the correspondence to Esterel it is possible to extend the tool to be able to incorporate models written in Esterel or an Esterel-based language such as ECL [11] as another model type.

We use a control skeleton generator based on work by Edwards [17] to obtain fast, efficient control code.

7. The control/dataflow interface

We now describe the control/dataflow interface, first for the simpler case of unit-rate and then treating the general case.

We call a model *unit-rate* if it can execute in a dataflow context by first consuming one value from each input port, and then producing one value on each output port. A dataflow graph can be unit-rate even if it contains non-unit-rate components, provided that a schedule exists that meets this condition at the boundaries (for example, we could have balancing decimation and interpolation blocks). Use of cyclo-static dataflow makes unit-rate dataflow graphs easier to achieve, even for cases where a model requires many samples to operate.

When a control model is embedded in a dataflow graph, it appears as a unit-rate model: one value is consumed by each input and one value is produced on each output. When a dataflow model (either a prim_model or a dataflow graph) is embedded in a control model, it may or may not be unit rate. Ports of the dataflow model can be bound either to ports or to signals belonging to the parent control model. In addition, parameters of the dataflow model can be bound to parameters, ports, or signals of the parent control model, except that structural parameters can only be bound to parameters of the parent. The streams seen by the dataflow model are produced by sampling the values of the input port bindings to produce values, and the values produced by the dataflow model are used to update the corresponding ports or signals of the control model. Values output by the dataflow model can be used for control (e.g. in an expression that causes a transition). If the embedded dataflow graph has unit rate at the boundary, the rates are by definition compatible with the rates of the parent control model.

It is frequently desirable to use a state machine to switch between dataflow systems that require multiple data values for each control step. To achieve this, we must tell the tool how many values each port of the hierarchical state will consume or produce. This is done by specifying an optional samples/step expression along with the port binding. If the expression is omitted, one value is moved (the unit rate case); otherwise it gives the number of values to move. However, if this feature is used, it imposes some restrictions.

If we use the movement expression, then the port of the child instance must be bound to a port, not to a signal, in the parent control model. Furthermore any port of the control model that is bound to a non-unit-rate port of some child may not be used in a condition or action on a transition, or in the inline action of an atomic state. The effect is that we have two kinds of ports: those that are only routed through to the child instances (which may be multirate), and those that can be used for control (which must be unit rate).

For gated models, the number of values read from and written to external ports is the same no matter which "side" of the gated model is executed. It is one (unit-rate) by default; if the user overrides this, both sides of the gated model must specify the same expression.

8. Modeling Example

In this section we present a simple example[1] to illustrate the various modeling constructs. The example is a simplified model of an automotive cruise controller. The top level of the design is modeled as a dataflow graph. CC_TOP is the main functionality. The rest of models (the environment) that create input stimuli and gather output for visualization have been omitted for simplicity. CC_TOP has **bool** inputs cc_on (cruise control on), cc_off (cruise

---

1. A number of more complex designs in multi-media and communication domains have been done. We have picked a tutorial one for brevity and simplicity.

control off), ig_on (ignition on), ig_off (ignition off), set (set cruising speed), resume (resume after pause), bpp (brake petal push) and **float** inputs acc (current acceleration), speed (current speed) and **float** output tvc (throttle valve control). The implementation of CC_TOP is an or-model denoted by top_or in the figure. Top_or has an atomic state S1 which is the start state and a hierarchical state CC. When ig_on is true the transition to CC is enabled and when ig_off is true CC_TOP reverts to the 'do nothing' state S1.

CC is implemented as an and-model with two concurrently executing *pages* Page1 and Page2 (a page is a user interface concept: a model that is drawn in the context of its parent). Page1 contains a gated-model that models the default behavior when the cruise control is turned off. This model is active only when the condition "cont == false" evaluates to true. When cruise control is on the gating condition evaluates to false and this model freezes retaining its internal state (if any). The internals of this model is a prim_model, normal_mode, which simply copies the acc input to the tvc output.

The model in Page2 that executes in parallel with page1 is again a hierarchical control model. This is an or-model cntr_fsm with two states S1 and Main. When cc_on is true the state Main becomes active. In the real design Main is a dataflow graph. In this example we have simplified to it to a single prim_model Control_Body. The code for Control_Body is

```
prim_model control_body {
    port in float speed;
    port out float tvc;
    port out bool cont;
    param float k1, k2;

    //local variables
    float speed_ref;
    bool snap_ref= true;
    float tvc_int;

    main_action {
        read(speed);
        if (snap_ref) {
            speed_ref = speed; snap_ref=false;
        }
        float delta = speed - speed_ref;
        tvc = tvc_int = k1 * delta + k2 * tvc_int;
        write(tvc);
        cont = true;
        write(cont);
    }
}
```

Control_Body essentially tries to bring the current speed of the vehicle to converge to the reference speed in a smooth manner. The parameters k1 and k2 control the manner in which this convergence happens. This behavior is illustrated in the output trace. Here the curve drawn with circles is the speed of the vehicle. At time 58 cruise control is turned on. At this point the reference speed is established and it can be seen that the vehicle speed converges to

Time In Seconds
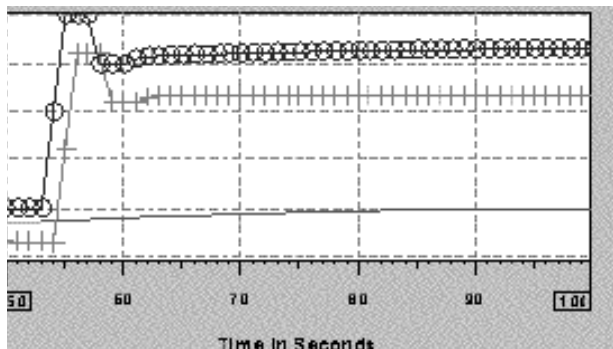
the reference speed. The second curve drawn with '+' is the tvc output. The curve at the bottom models the environment which for this time window is almost a constant.

This toy example illustrates the power and flexibility provided by the different modeling styles and the ability to nest models.

9. Conclusions

El Greco provides a powerful environment for modeling and validating the functionality of complex heterogeneous systems. The ability to model at different levels of abstraction combined with the ability to transform models from one level of abstraction to a lower level either by code generation or by manual refinement on a block by block basis all within the same environment, we believe, are key advantages. This combined with fast compiled simulations wherever possible makes El Greco an unique environment for rapid algorithm exploration.

10. Acknowledgments

References

[1] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems," *Int. Journal of Computer Simulation*, vol. 4, pp. 155-182, April 1994.

[2] G. Berry and G. Gonthier, "The Esterel synchronous programming language: Design, semantics, implementation," *Science of Computer Programming*, 1992, vol. 17, no 1, pp. 95-130.

[3] B. Lee and E. A. Lee, "Interaction of Finite State Machines with Concurrency Models," in *Proc. of 32nd Asilomar Conference on Signals, Systems, and Computers*, November 1998.

[4] C. Andre, "Representation and Analysis of Reactive Behaviors: A Synchronous Approach," *Proc. CESA '96*, IEEE-SMC, Lille, France, July 9-12, 1996.

[5] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, Vol. 8, No. 3, 1987, pp. 231-274.

[6] F. Belina, D. Hogiefe, A. Sarma, *SDL With Applications from Protocol Specification*, Prentice Hall International (UK), Hemel Hempstead, 1991.

[7] F. Balarin *et al.*, *Hardware-Software Co-design of Embedded Systems - The POLIS Experience*. Kluwer Academic Pub., 1997.

[8] P. Chou and G. Borriello, "Modal Processes: Towards Enhanced Retargetability through Control Composition of Distributed Embedded Systems," in *Proc. DAC 1998*, June 1998.

[9] T. Grötker, R. Schönen, and H. Meyr, "PCC: A Modeling Technique for Mixed Control/Data Flow Systems," in *Proc. European Design and Test Conference*, 1997.

[10] A. Seawright, U. Holtmann, W. Meyer, B. Pangrle, R. Verbrugghe, and J. T. Buck, "A System for Compiling and Debugging Structured Data Processing Controllers", *Proc.EuroDAC 1996*, September 1996.

[11] L. Lavagno and E. Sentovich, "ECL: A Specification Environment for System-Level Design", *Proc. DAC 1999*, June 1999.

[12] S. Liao, S. Tjiang, and R. Gupta, "An Efficient Implementation of Reactivity for Modeling Hardware in the Scenic Design Environment", *Proc. of 34th Design Automation Conf.*, June 1997.

[13] M. von der Beeck, "A Comparison of Statecharts Variants," *Proc. 3rd Int. Symposium on Formal Techniques in Real Time and Fault Tolerant Systems*, LNCS 863, pp 128-148, Springer Verlag, 1994.

[14] G. Bilsen, M. Engels, R. Lauwereins, J.A. Peperstraete, "Cyclo-Static Dataflow," *IEEE Trans. on Signal Processing*, Feb. 1996.

[15] J. Buck and E.A. Lee, Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model," *Proc. of IEEE Int. Conf. on Acoustics, Speech, and Signal Processing*, April 1993.

[16] G. Berry, "The Constructive Semantics of Pure Esterel", draft paper (http://www-sop.inria.fr/meije/esterel/esterel-eng.html), Jul. 1999.

[17] S. Edwards, "Compiling Esterel into Sequential Code", *Proc. 7th Int. Workshop on Hardware/Software Codesign (CODES-99)*, May 1999.