SC-DEVS: An efficient SystemC Extension for the DEVS Model of Computation

Felix Madlener, H. Gregor Molter, Sorin A. Huss
Integrated Circuits and Systems Lab
Technische Universität Darmstadt, Germany
{madlener|molter|huss}@iss.tu-darmstadt.de

Abstract

This paper describes a systematic approach to integrate the Discrete Event Specified System (DEVS) methodology into SystemC. It thus combines Model of Computation (MoC) specific properties and the features of an advanced SystemC environment. The execution of abstract system level DEVS models is comparable to pure SystemC models and is significantly faster compared to other DEVS environments. Thus, system level models based on abstract MoCs may easily be executed in a SystemC environment. The proposed integration is realized as a non-introspective extension to the SystemC 2.2 kernel. The DEVS models are implemented on an additional software layer above the SystemC simulation kernel. Our approach may be used simultaneously with other layered extensions, e.g., SystemC-AMS or TLM.

I. Introduction

In the recent past an increasing effort has been performed to extend the levels of abstraction and to meet the requirements of highly complex system level design. To establish such higher abstraction levels, Models of Computation (MoC) were introduced.

SystemC as a high-level modeling language aimed for the combined development of HW and SW and Transaction Level Modeling (TLM) as an extension, intended to model high-level system communication are widely known and have become a de facto standard for system level design. A lot of scientific work has been done and different commercial tools have been developed to improve an integrated design flow from a high-level SystemC design down to both, synthesizable HW and to embedable SW programs.

Nevertheless, the utilization of further MoCs is still important. SystemC itself has no fully formal semantics, which makes formal analysis as well as verification difficult. For this purpose, formally defined, abstract MoCs are more suitable. In certain scenarios domain specific MoCs are advisable. Scientific expertise can be reused by the employment of existing dedicated models.

In this work we describe a systematic approach to integrate a MoC into SystemC. This is performed by means of an additional software layer above the SystemC kernel which implements the target MoC. Using this approach the designer can benefit at the same time from both the model specific features and expertise and from the advanced SystemC design flow.

We selected the Discrete Event Specified System (DEVS) [13] as our target MoC. By being an event-based specification it is highly applicable for HW design. It is a well-known, formally defined MoC and a high effort of scientific research has been put into it.

We will show that it is possible to adapt the simulation kernel for the integrated MoC. This allows the reuse of the highly optimized SystemC simulation core and features an efficient execution of DEVS models.

This paper will review related work in Section II and then gives a short introduction into the formal specification of DEVS in Section III. In Section IV the implementation and integration of DEVS into SystemC is described. Section V gives some results which demonstrate the feasibility of the presented approach. We conclude this paper in Section VI.

II. Related Work

There is a lot of work available, which is aimed to overcome the lack of a formal SystemC specification and to reduce the need for other MoCs in this area. [12] summarizes formal methods which can applied to SystemC itself and [6] develops a formal MoC for a subset of the SystemC language.

[9] describes the integration of synchronous data flow models (SDF) in SystemC, [8] addresses the integration of a rule-based MoC (Bluespec). Compared to this work, these approaches exploit kernel level extensions. In contrast, it was our goal to keep the original SystemC kernel

978-3-9810801-5-5/DATE09 © 2009 EDAA

untouched and to utilize language level extensions only. [5] specifies an additional SystemC layer to ease the integration of different MoCs. The authors of [5] are applying both static and dynamic checks for MoC rules, while this approach enforces MoC rules by using SystemC and C++ language features.

An integration of Petri Nets is described in [10]. Each Petri Net is directly implemented as a SystemC module without any supporting framework, which makes this approach unusable for complex system specifications.

The integration of an analog MoC to support mixed-signal simulation is described in [1]. As it is focused on the analog simulation environment, it lacks a methodical description of the integration into SystemC execution environments.

III. Discrete Event Specified Systems – DEVS

The DEVS [13] formalism from Zeigler et al. is a great mathematical foundation for specifying hierarchically, concurrently executed, formal models. Being aware of the fact that the formalism covers both time-discrete and time-continuous models, we focus at the time-discrete ones only.

A DEVS system specification is a structure

$$\mathrm{S} = \langle \mathrm{Elems}, \mathrm{Conns}, \mathrm{Ports}_{\mathrm{in}}, \mathrm{Ports}_{\mathrm{out}}, \mathrm{tb} \rangle . \quad (1)$$

tb is the timebase, continuously or discrete, of the model. Elems is a set of hierarchically instantiated DEVS system specifications and/or atomic DEVS components. $\mathrm{Ports}_{\mathrm{in}}$ and $\mathrm{Ports}_{\mathrm{out}}$ are tuples $(p_1, \ldots, p_n)$ of unique port names for the input and output ports which receive or emit events.

Ports coupling is described by Conns, which is a set of tuples (i, o). i is an input port of $\mathrm{Ports}_{\mathrm{in}}$ or the output port from an instantiated submodule. Similarly, o is an output port of $\mathrm{Ports}_{\mathrm{out}}$ or an input port of Elems.

Each output port can be connected to multiple input ports, but no input may be linked to more than one output port. Every port has an associated set of event values including the $\diamond$ symbol representing the absence of an event.

An atomic DEVS component is described by the tuple

$$\mathrm{C} = (\mathrm{Ports}_{\mathrm{in}}, \mathrm{Ports}_{\mathrm{out}}, S, s_0, \delta_{\mathrm{int}}, \delta_{\mathrm{ext}}, \delta_{\mathrm{con}}, \lambda, \tau). \quad (2)$$

S is a non-empty set of states, with $s_0 \in S \cap \{\emptyset\}$ being the initial state of the DEVS component. Every state has an associated timeout $\tau : S \to \mathrm{tb}$.

Three different types of state transitions are defined:

Internal $\delta_{\mathrm{int}} : S \to S$: After the timeout $\tau(s)$ occurred, the component will do an internal state transition.

External $\delta_{\mathrm{ext}} : S \times \mathrm{Ports}_{\mathrm{in}} \times \mathrm{tb} \to S$: Iff the component receives an external event over one of it's input ports and no timeout occurs, it will do an external state transition $\delta_{\mathrm{ext}}(s, x, e)$, where $e < \tau(s)$ is the elapsed time since the last input event occurred.

Confluent $\delta_{\mathrm{con}} : S \times \mathrm{Ports}_{\mathrm{in}} \times \mathrm{tb} \to S$: Iff it receives an external event while an timeout occurs ($e = \tau(s)$), the next state will be computed by $\delta_{\mathrm{con}}(s, x, e)$.

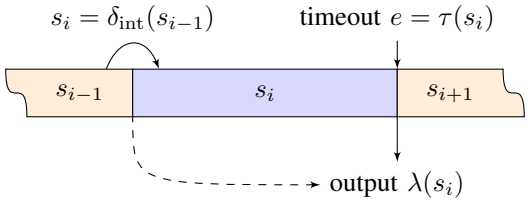

Fig. 1. State Transition and Output

Whenever a timeout $\tau(s)$ is hit, the component will emit output $\lambda(s)$, which can also be the absent event $\diamond$.

As depicted in Figure 1 an output occurs upon leaving a state, although its value is determined upon entering a state.

IV. SystemC Implementation

We now detail how a MoC, e.g., DEVS, can be integrated into SystemC. The implementation, hereinafter referred to as SC-DEVS, is realized as a non-introspective extension to the SystemC 2.2 kernel: the existing SystemC kernel must not be modified. Figure 2 depicts the relationship of the classes from the formal model and their SystemC counterparts.

The remainder of this section gives an in-depth view how the SystemC kernel is exploited to efficiently implement DEVS-specific classes and behavior. These classes form together a DEVS system as denoted in Equation 1.

A. Time

DEVS includes the explicit notation for an infinite amount of time, but SystemC does not: The time notation within SystemC is strictly discrete and finite. To include an infinite time notation and its behavior we extended the SystemC kernel by a `devs_time` class.

We can not derive the `devs_time` class directly from the normal `sc_time` class because of the normal C++ class downcasts. A `devs_time` object set to the infinite time may be automatically downcasted to a `sc_time` object *not* describing an infinite amount of time. Rather, we create the `devs_time` class to include a `sc_time` variable and a flag describing the additional symbol for the infinite amount of time. The standard arithmetic operators (+,−,⋆,/) were extended to handle mixed `sc_time` and `devs_time` arithmetic.

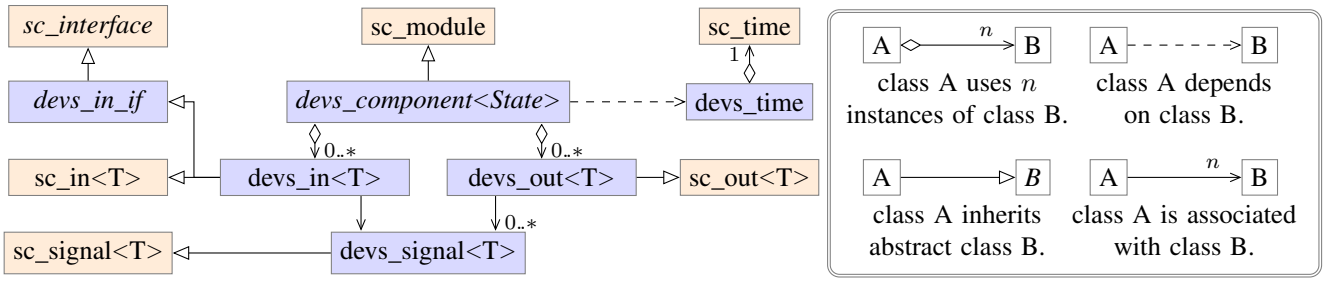

Fig. 2. UML Class Diagram for the SystemC DEVS Implementation

```
template<class State> class DEVS_Component : public sc_module { ... protected:
 virtual State initialize(void) const =0;
 virtual void output(const State& state) const =0;
 virtual devs_time time_advance(const State& state) const =0;
 virtual State external_transition(const State& state, const devs_time& elapsed) const =0;
 virtual State internal_transition(const State& state) const =0;
 virtual State confluence_transition(const State& state, const devs_time& elapsed) const =0; ... };
```

Fig. 3. Partial Class Definition of DEVS_Component

B. Model Specification

Atomic DEVS components are represented by the `devs_component<State>` class. Like in SystemC the ports of each component are described by the use of `devs_in` and `devs_out`. Connections between these ports are described by `devs_signal`.

Both the components and the signals are normally instantiated by the `sc_main()` function, but instantiation is not restricted to it. They may be also instantiated inside another component, thus giving the ability to model hierarchical designs.

1) Components: The atomic DEVS components are represented by classes derived from the abstract class `devs_component<State>`, which itself is derived from `sc_module`. To allow an easy reuse of existing models within SystemC, we map all elements of a formal DEVS specification directly into class member functions. The state space S of the component is implemented as a template parameter of the class. All functions for next state, output, and timeout are realized as pure-virtual, abstract class functions as listed in Figure 3. To implement an atomic DEVS component, only a state description must be supplied to the derived class as template parameter and realizations for the six pure-virtual, abstract functions have to be filled in.

2) Communication: In the DEVS MoC events on a signal reoccur, if the same "value" is written repeatedly to the output port. However, this is not supported by the original SystemC `sc_signal`. So, the class `devs_signal` capable of this feature was created. It is derived from `sc_signal` with the modified write methodology: If the same "value" reoccurs, it emits multiple events, i.e., one for each write operation.

Consequently, we had to develop dedicated port types to be used with this `devs_signal` class. These ports `devs_in` and `devs_out`, derived from `sc_in` and `sc_out`, can be coupled with `devs_signal`. By deriving DEVS ports and signals from the SystemC classes it is possible to exploit the elaboration comfort of SystemC, e.g., warnings for unbound ports.

All transition functions can query the `devs_in` ports to compute the next state. However, the transition functions must not write to the `devs_out` ports. This is performed exclusively by the `output` function, which is invoked every time after an timeout event occurred.

3) Transition Functions: The three functions for internal δ_{int}, external δ_{ext}, and confluence δ_{con} transitions of DEVS models are directly mapped to corresponding `C++` functions. The function definitions are depicted in Figure 3.

These functions may be executed in parallel as described later on in Section IV-C, but they are vulnerable to side-effects or even deadlocks. To remove the possibility that one encodes them to be vulnerable to side-effects, e.g., by using a shared object variable in multiple transition functions, they are declared `const`. Existing shared variables may be moved into the system state, which is synchronized by the simulation kernel.

However, we cannot guarantee that no side-effects are contained. One may use `mutable` object variables, which may be even *written* inside of the transition functions. Unfortunately, this is an intrinsic property of the `C++` language.

C. Elaboration and Simulation

Similar to SystemC each module has a two phase lifetime: elaboration and simulation.

The DEVS components have to react upon input port events. During the elaboration phase a special function is called which registers all input ports to the SystemC kernel. This information is used later by the components to react to received events.

In the simulation phase the model is executed. A normal simulation cycle is divided into four parts:

1) Determine whether the component was activated by timeout or by an external event.
2) Execute the required transition functions, calculating the follow-up state of the component.
3) Emit values over the output ports with the help of the `output` function, but only if necessary, e.g., when an internal transition event occurred.
4) Identify the timeout of the current state with the `time_advance` function and suspend until this point is reached in the simulation time or an event occurs.

The SystemC simulation methodology requires that all port outputs are stable within one delta-cycle. We mapped the four steps of the simulation cycle into a single SystemC delta-cycle. Therefore, we have a globally stable system at any point in time: All components compute simultaneously[1] their response in respect to the given inputs and their current states. The outputs are propagated to the sinks in the next simulation cycle, e.g., the next SystemC delta-cycle.

For further testing of the behavior of the models the developer may choose from three different simulation models, thus modifying the order and the calculation of transition functions in step 2 of the simulation cycle:

Needed Serial: Only the needed transition function is executed. The needed function can be determined in step 1 of the simulation cycle. This variant is supposed to give the best performance.

All Serial: All three transition functions, δ_{int}, δ_{ext}, and δ_{con}, are executed in a serial, but randomized order. The follow-up states of the two inactive transitions are discarded.

Parallel: All three transition functions are executed in parallel with the help of `pthreads`. Again, only the follow-up state of the active transition is kept.

Assuming a correct DEVS model, all three variants will show the same behavior. However, in case of an incorrect model, the behavior might differ. This difference can be used to locate and eliminate the modeling error.

[1] Simultaneously computation is serialized by the SystemC kernel.

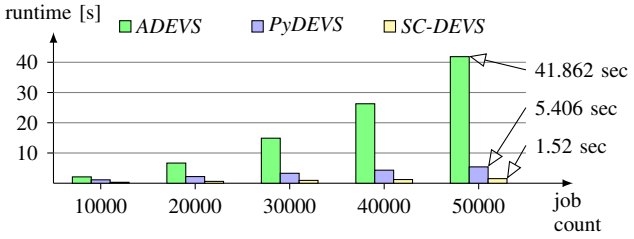

Fig. 4. Simulation Effort for the *gpt* Model

V. Results

To demonstrate the feasibility and performance of our approach to map DEVS models on top of SystemC, we evaluated two different scenarios. First, we implemented a small DEVS model and compared our simulator with other existing DEVS simulators. In a second scenario we have developed a more complex model of a driving assistant application which was implemented both in DEVS and in pure SystemC. All simulations were run on an Intel Core2 Quad (2.4 GHz) processor with Debian 4.0 (Etch) Linux.

A. *gpt* Example

There is a great amount of DEVS simulators available [4] based upon different implementation technologies, e.g., *PyDEVS* [2] (Python), *DEVSJAVA* [11] (Java), or *ADEVS* [7] (`C++`). We compared our implementation to *ADEVS* and *PyDEVS* running the *gpt* example from *DEVSJAVA*. By this, we show that the DEVS layer above the SystemC core has no substantial overhead compared to designated DEVS-only simulators. A really fair comparison with *DEVSJAVA* is not possible, as it is GUI-based and does not include any timing measurement methods.

The *gpt* example consists of three components: a job generator, a processor, and a transducer. The generator creates jobs, e.g., events, which are to be consumed by the processor. The transducer analyzes the processing rate and controls the job generation.

Figure 4 shows the resulting simulation effort. Obviously, the proposed SystemC DEVS simulator performs best with a simulation time of 1.52 sec. The main reason for this property is the optimized SystemC simulation kernel, which is more advanced than in most other simulators. The figures prove that the overall performance gain obtained by using the optimized SystemC simulation kernel clearly outweighs any overhead caused by an additional MoC software layer. This will be true for many other MoCs as well, where no highly optimized simulator exists.

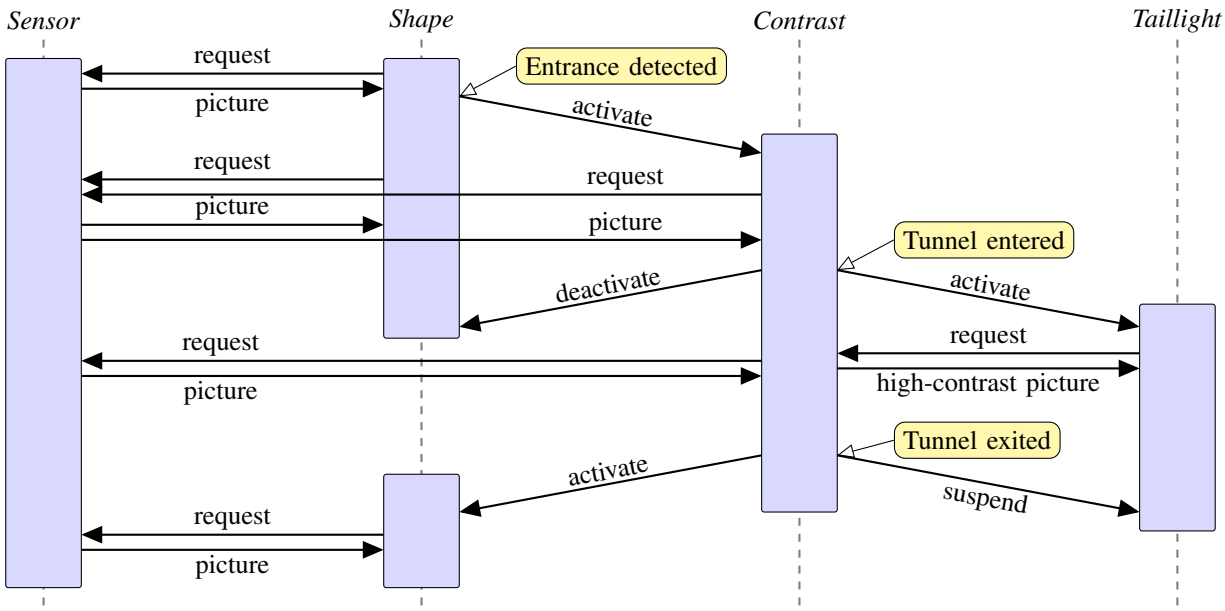

Fig. 5. Object Diagram of the AutoVision Example

B. AutoVision Example

For a detailed comparison between our SC-DEVS and pure SystemC models we used a more complex application, which is based on the AutoVision example presented in [3]. It consists of several distinct components for vision enhancement and for automated object recognition aimed to a driving assistance scenario. Figure 5 gives an overview over the following components and their interaction:

Sensor: Provides pictures to all requesting components.

Shape: Performs picture requests and scans the result for important shapes (e.g. other cars). If the shape of a tunnel entrance is found, then the *Contrast* component is invoked.

Contrast: Enhances the *Sensor* picture and recognizes, when the car enters or leaves a tunnel, in which case it activates or suspends other components.

Taillight: Provides object information to the driver based on taillight traces. It operates when the car is inside the tunnel where it is too dark for the *Shape* component to operate properly.

All four components run concurrently allowing for the evaluation of the different parallel execution models our simulator provides. By using different request intervals for *Shape* and *Contrast* the situation of multiple competing requests arriving at the same time can be modeled. This allows the in-depth analysis of concurrent communication.

The performance evaluation focuses on two different characteristics: Overall simulation runtime (throughput) and processing time for a single event (latency).

To get a fair comparison between SC-DEVS and SystemC models we need to differ between state transitions with low and high processing time. Measurements with low processing times give us a more realistic view how efficient, in terms of throughput, SC-DEVS is, when compared to the raw SystemC implementation. High processing times measure the latency of each execution semantic (i.e., *Needed Serial*, *All Serial*, and *Parallel*) and give us the ability to compare them to each other.

The SystemC model is a reimplementation of the DEVS model with an equivalent behavior. The three transition functions δ_{int}, δ_{ext} and δ_{con} are implemented within a *SC_THREAD*. The selection of the appropriate transition function and the time advancement, which is handled transparently to the developer by the DEVS kernel, had to be reimplemented manually within each of the SystemC models.

Figure 6 depicts the results of the AutoVision example. The graph on the left hand side visualizes the kernel performance for an increasing number of events. All variants scale linearly with the number of events. While *Needed Serial* and *All Serial* show a similar performance compared to the SystemC simulator, the *Parallel* simulation is approximately 3 times slower (33.3 seconds compared to 11.2 seconds). This stems from the additional kernel overhead for thread handling and synchronization.

The right hand side of Figure 6 depicts the situation for transitions with high processing times. Compared to

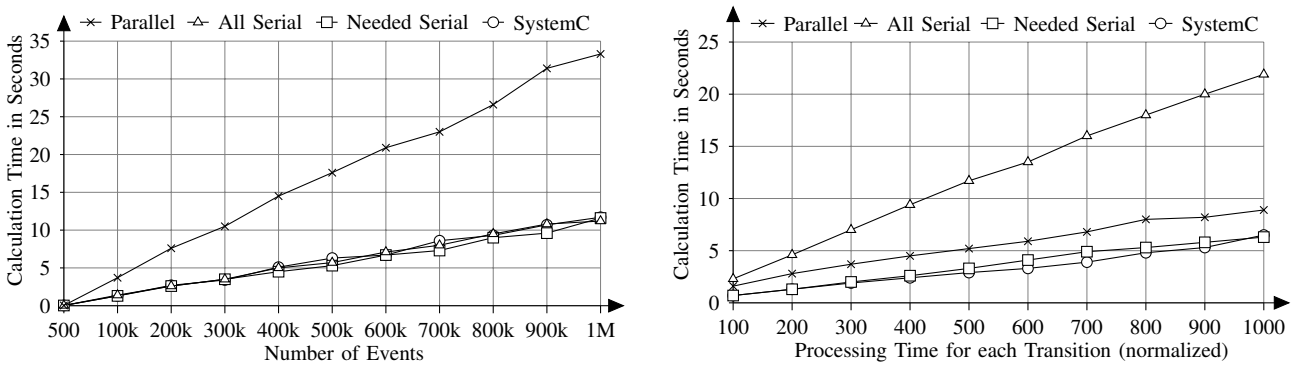

Fig. 6. Simulation Results for the AutoVision Example

the *All Serial* execution, the *Parallel* execution is 3 times faster as it can calculate the three transition functions in parallel.

The *Needed Serial* simulator option again shows a similar performance to the pure SystemC simulator. This has been expected as this simulator variant realizes a similar behavior to the original SystemC kernel. While the *Needed Serial* is slightly faster than the *Parallel* execution model, it can not detect design flaws which may originate from synchronization or concurrency issues. The *Parallel* model is able to detect them and did it multiple times throughout our tests.

VI. Conclusion

We have presented a systematic approach to integrate the Discrete Event Specified System model into SystemC and demonstrated its feasibility. We have extended the elaboration and simulation kernel of SystemC by an additional layer to support DEVS models. The implementation provides different execution modes to process the state transition functions of a model. As expected, the *Needed Serial* gives the best overall results for the different DEVS simulator variants.

We demonstrated a certain tradeoff between simulation overhead and the runtime of the state transitions. If the transition functions require a noteworthy processing time, then the usage of the *Parallel* execution model becomes interesting. It does not slow down the simulation too much and it can give additional insights into the model behavior.

A complex DEVS model has been compared with an equivalent SystemC model. The results show that the overhead of an additional software layer is negligible and the performance is similar to the pure SystemC model. Thus, MoCs featuring considerably higher abstraction levels may be introduced to a SystemC-based design flow at almost no overhead at all.

References

[1] H. Aljunaid and T. Kazmierski, "SEAMS - a SystemC environment with analog and mixed-signal extensions," in *Proc. of the Intl. Symposium on Circuits and Systems ISCAS '04*, vol. 5, May 2004, pp. 281–284.

[2] J. Bolduc and H. Vangheluwe, "A Modeling and Simulation Package for Classic Hierarchical DEVS," MSDL, School of Computer McGill University, Tech. Rep., 2002.

[3] C. Claus, W. Stechele, and A. Herkersdorf, "Autovision - A Runtime Reconfigurable MPSoC Architecture for Future Driver Assistance Systems," *it - Information Technology*, vol. 49, no. 3, pp. 181–186, 2007.

[4] DEVS Standardization Group, "DEVS tools," http://www.sce.carleton.ca/faculty/wainer/standard/ [Accessed: Sep. 05, 2008].

[5] F. Herrera and E. Villar, "A framework for heterogeneous specification and design of electronic embedded systems in SystemC," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 12, no. 3, pp. 1–31, 2007.

[6] K. Man, "SystemC^{FL}: A formalism for hardware/software codesign," in *Proc. of the 2005 European Conf. on Circuit Theory and Design*, vol. 1, Aug. 2005, pp. 193–196.

[7] J. Nutaro, "ADEVS (A Discrete EVent System simulator) C++ library," http://www.ornl.gov/~1qn/adevs/ [Accessed: Sep. 05, 2008].

[8] H. Patel, S. Shukla, E. Mednick, and R. Nikhil, "A rule-based model of computation for SystemC: integrating SystemC and Bluespec for co-design," in *Proc. of the ACM and IEEE Intl. Conf. on Formal Methods and Models for Co-Design, MEMOCODE '06.*, Jul. 2006, pp. 39–48.

[9] H. D. Patel and S. K. Shukla, "Towards a heterogeneous simulation kernel for system-level models: A SystemC kernel for synchronous data flow models," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 24, no. 8, pp. 1261–1271, 2005.

[10] C. Rust, A. Rettberg, and K. Gossens, "From high-level Petri nets to SystemC," *IEEE Intl. Conf. on Systems, Man and Cybernetics 2003*, vol. 2, pp. 1032–1038, Oct. 2003.

[11] H. S. Sarjoughian and B. P. Zeigler, "DEVSJAVA," http://www.acims.arizona.edu/SOFTWARE/software.shtml [Accessed: Sep. 05, 2008].

[12] M. Y. Vardi, "Formal Techniques for SystemC Verification; Position Paper," in *Proc. of Design Automation Conf., DAC '07*, 2007, pp. 188–192.

[13] B. P. Zeigler, T. G. Kim, and H. Praehofer, *Theory of Modeling and Simulation*. Academic Press, Inc., 2000.