Heterogeneous Behavioral Hierarchy Extensions for SystemC

Hiren D. Patel, *Student Member, IEEE,* Sandeep K. Shukla, *Senior Member, IEEE,* and Reinaldo A. Bergamaschi, *Fellow, IEEE*

***Abstract*—System level design methodology and language support for high-level modeling enhances productivity for designing complex embedded systems. For an effective methodology, efficiency of simulation and a sound refinement-based implementation path are also necessary. Although some of the recent system level design languages (SLDLs) such as SystemC, SystemVerilog, or SpecC have features for system level abstractions, several essential ingredients are missing from these. We consider: 1) explicit support for multiple models of computation (MoCs) or heterogeneity so that distributed reactive embedded systems with hardware and software components can be easily modeled; 2) the ability to build complex behaviors by hierarchically composing simpler behaviors and the ability to distinguish between structural and heterogeneous behavioral hierarchy; and 3) hierarchical composition of behaviors that belong to distinct MoCs, as essential for successful SLDLs. One important requirement for such an SLDL should be that the simulation semantics are compositional, and hence no flattening of hierarchically composed behaviors are needed for simulation. In this paper, we show how we designed SystemC extensions to facilitates for heterogeneous behavioral hierarchy, compositional simulation semantics, and a simulation kernel that shows up to 40% more efficient than standard SystemC simulation.**

***Index Terms*—Behavioral decomposition, behavioral modeling, embedded system design, heterogeneous behavioral hierarchy, hierarchical finite state machine (HFSM), hierarchical synchronous data flow (SDF), models of computation (MoCs), simulation efficiency, structural modeling, system level designs, SystemC.**

I. Introduction

VARIOUS system level design languages (SLDLs) have been proposed for the specification and modeling of complex and heterogeneous hardware and software systems. SLDLs such as SpecC [1], SystemVerilog [2], and SystemC [3] provide system-level abstractions for hardware modeling and to a lesser extent for software modeling. Environments like Ptolemy II [4] and Metropolis [5] address embedded software design and/or platform-based system design. Ptolemy II uses hierarchical heterogeneity to model designs, whereas Metropolis uses a refinement-based methodology to express their designs. These SLDLs have their own limitations such as not being able to represent the model in register-transfer language (RTL) for Ptolemy II and representing heterogeneous hierarchy in Metropolis. These restrict their widespread use in industry for modeling heterogeneous hardware and software systems. Each of these SLDLs and their supporting frameworks have certain defining characteristics as well as shortcomings, which need to be addressed for making them successful in the embedded system design community. On top of these limitations, there are certain industrial trends that gate their success path. SystemC has been gaining traction in the industry, and rendering SystemC with capabilities necessary for a successful SLDL is essential for this trend to continue. However, SystemC was originally developed based on discrete-event (DE) simulation semantics (similarly to Very-High-Speed Integrated Circuit Hardware Description Language (VHDL) and Verilog). The DE semantics do not lend themselves easily to the modeling of fully heterogeneous systems. This has the detrimental effect of either restricting the types of designs that can be modeled using SystemC or imposing a significant overhead in code size and simulation speed, if such designs are modeled using the current SystemC implementation. As a result, we add simulation semantics distinct from the DE semantics to support the features we deem necessary.

From our experience with various SLDLs, we have identified three important characteristics for system level modeling. These are: 1) support for multiple models of computation (MoCs) or heterogeneity in the same language; 2) ability to build complex behaviors by hierarchically composing simpler behaviors, coupled with the ability to distinguish between structural and heterogeneous behavioral hierarchy; and 3) the ability to compose behaviors hierarchically that belong to distinct MoCs for exploiting heterogeneity and hierarchy.

This third feature allows for intuitive modeling and efficient simulation of complex system level models. In fact, a lot of embedded system models today are heterogeneous and best built compositionally from smaller components whose behaviors are governed by distinct MoCs. Furthermore, the simulation semantics for such SLDLs should also be compositional so that flattening of hierarchically composed behaviors is not needed for simulation. This paper presents methods and implementation of these characteristics as extensions to the SystemC kernel, to allow it to natively model heterogeneous behavioral hierarchy and at the same time achieve simulation

Manuscript received December 6, 2005; revised February 14, 2006. This work was supported in part by the National Science Foundation under Project CCR-0237947 and in part by the Semiconductor Research Corporation project. This paper was recommended by Associate Editor R. Camposano.

H. D. Patel and S. K. Shukla are with the Electrical and Computer Engineering Department, Virginia Polytechnic and State University, Blacksburg, VA 24061 USA (e-mail: shukla@vt.edu).

R. A. Bergamaschi was with the Electrical and Computer Engineering Department, Virginia Polytechnic and State University, Blacksburg, VA 24061 USA. He is now with the IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA.

Digital Object Identifier 10.1109/TCAD.2006.884859

speeds up to 40% faster than standard SystemC simulation of the same design. This paper significantly extends our previous work on heterogeneous simulation kernel extensions for SystemC [6]–[8].

II. Why Heterogeneity, Behavioral Hierarchy, and Heterogeneous Behavioral Hierarchy

The capability for heterogeneity is the first extension we provide by introducing interoperable MoCs with SystemC [7], [9], [10]. The capability of heterogeneity in any SLDL promotes design decomposition based on the inherent behaviors of the components, which we term behavioral decomposition. Design decomposition is the process of categorizing a design into its most basic elements. On the other hand, behavioral decomposition does the same except the basic elements are governed by particular MoCs. Furthermore, large system models comprised of small behavioral subcomponents may consist of smaller components embedded within the subcomponents. This leads us to behavioral hierarchy [8] as the second extension of SystemC that we promote. Behavioral hierarchy is the ability to build a design by composing smaller behaviors hierarchically. Our extensions to SystemC support behavioral hierarchy encouraging designers to seamlessly compose behaviors without having to worry about the semantics that define the composition. This is because we define compositional execution semantics for hierarchical compositions [8] also implemented in the extended SystemC kernel. Finally, we define heterogeneous behavioral hierarchy as the ability to hierarchically compose smaller behaviors belonging to distinct MoCs. With heterogeneous behavioral hierarchy, a designer can build system level models with hardware/software components that may be modeled with a hierarchical controller using the hierarchical finite state machine (HFSM) MoC, a digital signal processing (DSP) core with the synchronous data flow (SDF) MoC, a computationally intense component within the DSP core following the continuous-time (CT) MoC and so on.

Further elaborating, structural hierarchy is an encapsulation technique used during modeling, where subcomponents are embedded within other components and communicate through their ports. For example, a four-bit adder may consist of four one-bit adders (modeled as subcomponents) where at first, a one-bit adder is modeled at the RTL abstraction level, after which four instances of one-bit adders are connected with extra glue logic to represent a four-bit adder. This is an example of structural hierarchy where the modeling becomes manageable. On the other hand, behavioral hierarchy is the ability to analyze a design by decomposing it into small behaviors and then composing these behaviors hierarchically to complete the design. The behaviors in our approach are realized by specific MoCs.

Most widely used SLDLs such as SystemVerilog and SystemC allow for structural hierarchy but lack support for behavioral hierarchy. Taking SystemC as our SLDL in focus, we see that it provides structural hierarchy through C++ composition techniques, where an `SC_MODULE` contains other `SC_MODULE` instances and this embedding can go to any depth. However, the behaviors encapsulated inside these embedded modules could be `SC_THREAD` or `SC_METHOD` which are simulated only using DE semantics. Furthermore, if a module embeds another behavior, then it must be modeled in a different module to represent the hierarchy of one being embedded in another. Moreover, during simulation, all these structurally embedded processes are treated at the same level thereby flattening any implied hierarchy. This loses out on the extra information provided by the hierarchy and the corresponding abstraction. This means that with respect to simulation, structural hierarchy provides no benefit, but for modeling purposes it provides a good abstraction mechanism. Behavioral hierarchy, as we see it, provides benefits during modeling and simulation. Behavioral decomposition of the design provides encapsulation and abstractions during modeling; whereas for simulation, the proposed simulation kernel extensions operate on only that level of hierarchy. For example, if an FSM has an embedded FSM subsystem within one of its state, then a separate instance of the FSM simulation kernel executes the subsystem. We contend that behavioral hierarchy is a key in not only increasing the modeling fidelity [9] but also aiding in simulation efficiency. Also note that behavioral hierarchy is not totally independent of structural hierarchy, except that our design's behaviors at the different levels of hierarchy may be governed by different MoCs.

In particular, we present our multi-MoC hierarchical SystemC extensions for the FSM and SDF MoCs, which are interoperable with the SystemC DE simulation kernel to support heterogeneous behavioral hierarchy. Our kernels implement the execution semantics for these MoCs and not the communication. We select HFSM and SDF MoCs only as examples of some of the many MoCs that may be suitable for hardware modeling. However, this does not mean that they are the most suitable for all hardware models or other applications, but serve well as a proof of concept for heterogeneous behavioral hierarchy in SystemC. We demonstrate the modeling paradigm with two examples. The first one is a polygon in-fill processor (PIP) example and the second is an abstract power state model example. The power state example is of a hierarchical FSM and the PIP exhibits heterogeneous behavioral hierarchy. These nontrivial examples show that behavioral decomposition is a good means of dissecting complex designs into small behaviors and then composing the overall model hierarchically. Hence, aside from the advantages of system level abstraction and well-defined simulation semantics, we also demonstrate advantages in simulation efficiency. We report approximately 40% simulation improvement over an analogous implementation of the PIP in standard SystemC.

A. Main Contributions

The main contributions of this paper are: 1) rendering current industry standard SystemC with the ability of modeling heterogeneous behavioral hierarchy; 2) providing compositional simulation semantics for heterogeneous behavioral hierarchy so as to preserve the hierarchy during simulation of complex system models without flattening the hierarchy; 3) demonstrating how to model with the extended SystemC using examples of a

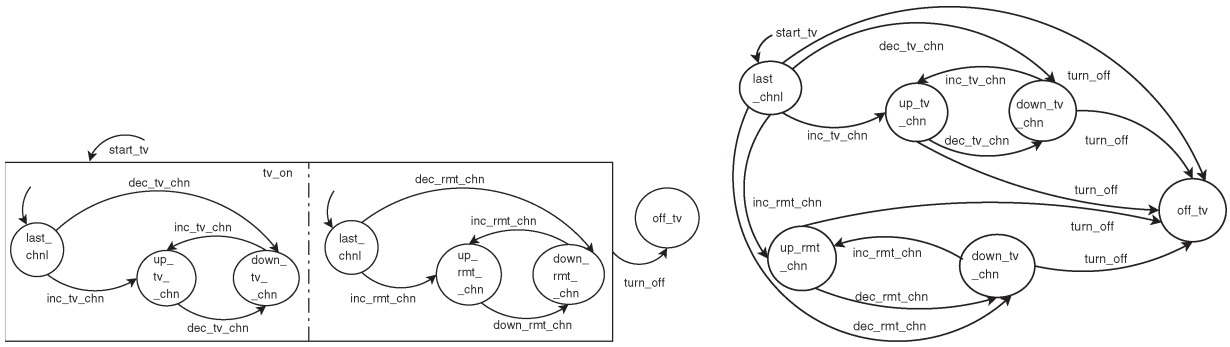


Fig. 1. State space abstraction (a) hierarchical FSM and (b) equivalent flat FSM.

graphics PIP and a power state model, and perform simulation experiments with our implementation of the simulation kernel extension; and 4) showing the benefit of heterogeneous behavioral hierarchy in terms of simulation efficiency. In particular, we show up to 40% simulation efficiency enhancement with our first-cut implementation of the new simulation kernels.

B. Organization

The remainder of this paper is organized as follows. In Section III we present the background regarding MoCs. This is followed by discussions on behavioral hierarchy, behavioral modeling, structural hierarchy, and structural modeling. Using simple examples we describe these terms. In Section IV, we describe and distinguish this paper from other popular heterogeneous system level design environments. Section V introduces the simulation semantics for behavioral hierarchy particularly focusing on the hierarchical FSMs followed by its implementation in Section VI. We present two examples of the PIP and the abstract power state example in Section VIII and conclude in Section IX.

III. Background

A. Fidelity and Multiple MoCs

An MoC describes the manner in which the communication and the computation occur in that particular MoC. There are additional semantics that discuss the interaction between varying MoCs as well as nesting of components modeling different MoCs such as those in Ptolemy II [4]. Fidelity [9] is the capability of the framework to faithfully model a theoretical model of computation. Our first contribution to increasing modeling fidelity in SLDLs is shown in SystemC-H [6], [11], [12], which promotes the idea of heterogeneity in SystemC via the introduction of MoC-based kernel extensions. Heterogeneity, in turn, facilitates decomposition of the design according to the behavior of the components such that the design can be realized as communicating subcomponents, each following a specific MoC. This leads us onto another key and natural abstraction of heterogeneous behavioral hierarchy, which we discuss after briefly introducing the two MoC extensions we provide in this paper.

1) SDF MOC: The SDF MoC is a subset of the data flow paradigm which dictates that a program may be divided into data path arcs and computation blocks. The program is then represented as a directed graph connecting the computation blocks with the data path arcs. A model following the SDF MoC can be statically scheduled for execution [7], [13], [14].

2) FSM MoC: We employ the Starcharts semantics [15] which emphasizes the importance of separating the concurrence MoC from the FSM semantics. The main contribution of Starcharts is in decoupling the concurrence MoC from the FSM semantics with the understanding that a concurrence model can be combined with the Starcharts FSM to express the desired behavior. Starcharts show how to embed HFSMs and concurrence MoCs at any level in the hierarchy. The hierarchy can be of arbitrary depth and Starcharts also provides the semantics required for it to interact with other MoCs. Starcharts are depicted as directed graphs, where the vertices represent the states and the edges represent the transitions. Every transition is defined as a guard-action pair. The action is only fired once the guard evaluates to true. Once the transition is enabled the state of the FSM moves to the destination state of the enabled transition. The two types of actions are `choice` and `commit` actions. Choice actions execute whenever the corresponding transition is enabled and commit actions only execute before the state change. This is to allow multiple executions of an FSM without changing the state to produce an output via the choice actions. Fig. 1 shows an example of a television button control FSM. A `reaction` of an FSM is a triggering of one enabled transition. A state or transition is said to contain a `refinement` if it has a component embedded within it. The refinement is interchangeably termed as `slave` subsystem whereas the container is the `master` system.

B. Behavioral Hierarchy

To grasp a better understanding of behavioral hierarchy, let us look at a simple television control (TV) example in Fig. 1. Fig. 1(a) displays a hierarchical version of the TV example and Fig. 1(b) shows its corresponding flat FSM. Once the television is turned on, channels can be flipped either using the television itself (named with `_tv_`) or the remote device (named with `_rmt_`). Note that this example has a different kind of behavioral hierarchy that employs AND concurrence in

FSMs [16]. The AND concurrence allows for orthogonal FSMs to execute together. Therefore, once the transition `start_tv` is taken, both the orthogonal FSMs separated by the dotted line execute and enter the `last_chnl` state. In Fig. 1(a), the `tv_on` state encapsulates all behavior of changing the channels via the remote and the television. Also, notice that only one transition to `off_tv` is required from this state, whereas in Fig. 1(b) every state must have a transition to turn the television off. Furthermore, the master FSM in Fig. 1(a) only sees two states, `tv_on` and `off_tv`. In comparison, the flat version of the model appears more complex due to the lack of any hierarchy.

Regarding simulation for behavioral hierarchy, each level of hierarchy is simulated by an instance of that MoC's scheduler. Fig. 1(a) uses three instances of an FSM scheduler to simulate the model. One instance simulates the two states in the master FSM and the other two are invoked for each of the embedded FSMs in the AND state `tv_on`. Hence, abstracting the state space effectively.

This shows two advantages of behavioral hierarchy for state space abstraction. The first is during modeling and the second is during simulation. For modeling, the idea is similar to structural hierarchy in which design decomposition helps in categorizing large designs, except with the difference that behavioral decomposition realizes the basic elements as MoCs. For simulation, the behavioral hierarchy is preserved by multiple instances of kernels being used to simulate each level of hierarchy, hence not flattening the hierarchy.

C. Behavioral Modeling Versus Structural Modeling

To clarify common confusion, we distinguish between behavioral and structural modeling as two different techniques for modeling. Behavioral modeling refers to design decomposition based on their behaviors such that the resulting design describes a sequence, composition, and hierarchy of behaviors that are realized by MoCs. Popular SLDLs such as SystemVerilog and SystemC allow for design decomposition but lack in behavioral hierarchical modeling.

In other words, they primarily support structural modeling, which also promotes design decomposition, except that the composition of basic behavioral elements is through ports and modules. For example, a basic element may be modeled in terms of registers, AND/OR/NOT gates, wires, logic expressions, and assignments. This basic element can then be composed structurally via ports and modules. This is what we term structural modeling. Now, when these components are composed to form a hierarchy, they exercise structural hierarchy. In fact, structural hierarchy and behavioral hierarchy are not completely orthogonal to each other. By using structural hierarchy, we lose out on the two main advantages obtained from using behavioral hierarchy. The first disadvantage is that simple behaviors represented by MoCs cannot be composed hierarchically to express composite behaviors in order to build a behavioral model. The second is that since structural hierarchy is only a manner of encapsulation during modeling, the SystemC simulation scheduler flattens the structural hierarchy not taking advantage of the state space abstraction through hierarchy. On the other hand, behavioral hierarchy promotes structural hierarchy except that the basic elements follow varying MoCs. With the understanding that one SystemC process expresses a particular behavior, each state in Fig. 1(a) models a behavior, although it may be simple in this example. Then, during simulation, SystemC expands the hierarchical composition of Fig. 1(a) into eight processes where `tv_on` is also a SystemC process encapsulating the other processes. The flattened version of the FSM in Fig. 1(b) requires six SystemC processes with numerous additional transitions. A designer may choose to model the entire FSM in one SystemC process for this particular example, but for more complex designs where each state models a large and complex behavior on its own, it is conceivable that one SystemC process models each state of a particular FSM. This shows that structural hierarchy breaks the state space abstraction with respect to simulation. However, not employing structural hierarchy makes components less reusable, difficult to manage, and reduces the clarity of the model. Unfortunately, most system level design frameworks such as Verilog [17] or VHDL [18] only support a homogeneous modeling environment. As a result, targeting MoCs that express FSM, SDF, etc., or some behavioral hierarchy composition of them, have to be mapped to the DE semantics of most HDLs, which degrades the simulation performance along with requiring a certain level of expertise in being able to programmatically model the MoCs using DE. This makes behavioral hierarchy an attractive alternative.

D. Extending SystemC's Kernel

Our experience in augmenting SystemC [3] with capabilities for heterogeneity and heterogeneous behavioral hierarchy exposed limitations with SystemC's kernel. These limitations make it difficult for us to introduce heterogeneity across all the MoCs, especially with SystemC's reference implementation of the DE MoC. In order to understand the limitations SystemC's DE MoC imposes on our extensions, we first briefly describe SystemC DE kernel's singleton pattern implementation [19]. Note that SystemC's DE kernel is one implementation of the DE semantics and there can be various different implementations. However, current SystemC's implementation [7]–[9], [11], [20] makes it difficult for users to extend the kernel

Listing 1. `static` declaration of simulation Context

```
1 static sc_simcontext* sc_curr_simcontext = 0.
```

SystemC [3] emerged as an open-source hardware description language implemented in C++. Its current implementation (version 2.1) is geared toward modeling at RTL and abstraction levels beyond RTL. The C++ language provides SystemC with the capabilities of classes, objects, polymorphism, inheritance and templates - all very powerful techniques for encapsulation and abstractions. SystemC's reference implementation adheres to DE MoC, following an evaluate-update paradigm [3], [9]:

Listing 2. `sc_get_curr_simcontext()` implementation

```
1 sc_simcontext * sc_get_curr_simcontext()
2 {
3   if(sc_curr_simcontext == 0){
```

```
4 #ifdef PURIFY
5     static sc_simcontext
          sc_default_global_context;
6     sc_curr_simcontext = &
          sc_default_global_context;
7 #else
8   static sc_context*
          sc_default_global_context = new;
        sc_simcontext;
9   sc_curr_simcontext =
        sc_default_global_context;
10 #endif
11     }
12   return sc_curr_simcontext;
13 }
```

SystemC requires the user to associate the modeled behavior with particular SystemC processes. The types of SystemC processes are SC_METHOD and SC_THREAD, which are implemented with varying thread packages depending on the operating system and the version of operating system. These thread packages are wrapped into a coroutine package that provide interfaces for starting, terminating, switching, etc., the threads. The class responsible for simulating the model is implemented in sc_simcontext, which interacts with the coroutine classes. The sc_simcontext when starting the simulation creates a static instance of sc_simcontext which is used as the default context. Listing 1 shows the declaration of the variable responsible for holding the current simulation context being declared as static. This variable is assigned its static object in sc_get_curr_simcontext() shown in Listing 2, which is invoked from the main entry function call sc_start(), shown in Listing 3. The static nature of sc_simcontext restricts another instance from replacing the default context, thus making SystemC's DE kernel follow a singleton design pattern [9]

Listing 3. sc_start () implementation

```
1 void sc_start(const sc_time& duration)
2 {
3   sc_get_curr_simcontext ()->simulate (duration);
4 }
```

We do not alter the existing SystemC reference implementation, hence the singleton pattern design of SystemC's DE kernel imposes hard and fast restrictions on how we introduce heterogeneity with SystemC. For example, it is not possible to have two concurrently executing instances of SystemC's DE kernel. Even though the program code for simulation is tidily encapsulated in class sc_simcontext, two instantiations of sc_simcontext only results in loss of context information. However, we require the capability of having multiple instances of the DE kernel such that one instance may be embedded within another, each instance may further be embedded in other MoCs, separate instances may have different temporal properties, and so on.

Another artifact of SystemC kernel's implementation is that all SystemC processes are treated at the same level of hierarchy, in essence flattening whatever hierarchical structure presented in the model [8], [20]. The SystemC processes as seen in class sc_simcontext, are stored in a list data structure. The simulation kernel then executes these SystemC processes by iterating through a list that contains all the ready-to-run SystemC processes. Therefore, hierarchical decomposition of a model during simulation is completely flattened. This makes nesting DE component within other domains such as FSM and SDF problematic. Ideally, we would want a separate instance of the DE kernel to simulate the nested DE component and a different instance for the top level DE components.

As of now, we cannot explore behavioral hierarchy with SystemC's DE kernel, but our future work is to investigate these necessary hierarchical compositions. We plan to implement a secondary DE kernel that reuses several SystemC data structures, data-types and channels as well as the threading packages. However, we will change the design to allow nesting of DE inside other MoCs and exploring temporal synchronization of different DE components, multiple instances of DE components at varying depths of the hierarchy and embedding of DE components within other MoCs.

IV. Related Work

There are several projects that employ the idea of MoCs as underlying mechanisms for describing behaviors. Examples of such design frameworks and languages are Ptolemy II [4], Metropolis [5], and SystemC-H [12]. We briefly describe some of the popular design frameworks employing MoCs and discuss how our attempt at integrating MoCs with SystemC differs from those of these design frameworks.

A. Ptolemy II

A promoter of heterogeneous and hierarchical designs for embedded systems is the Ptolemy II group at University of California (UC), Berkeley [4]. Even though Ptolemy II is one of the popular design environments that support heterogeneous behavioral hierarchy, it has its own limitations. First, Ptolemy II is geared toward the embedded software and software synthesis community and not targeted toward hardware and SoC designers. Therefore, certain useful semantics that may be considered essential for hardware modeling may not be easily expressible in Ptolemy II [8]. The Starcharts semantics do not allow a run-to-complete such that the particular FSM refinement continues execution until it reaches its final state before returning control to its master model. A designer must incorporate a concurrence mechanism to allow for this in Ptolemy II. We see this as an important characteristic useful in treating the refinement as an abstraction of functionality, which requires director extensions in Ptolemy II. Another crucial factor is that designers often need a single SLDL or framework to model different levels of abstractions from system level to RTL. In a framework such as Ptolemy II, RTL descriptions are not possible. However, since our extensions are interoperable with SystemC, models using our extensions may be refined with components exhibiting different levels of abstraction. Lastly, there is a large IP base built using C/C++ that is easier to integrate with an SLDL built using C/C++, which is not the case with Ptolemy II.

B. SystemC-H

An experimental prototype developed to study heterogeneity for SystemC labeled SystemC-H [12] introduces three MoCs interoperable with SystemC 2.0.1. These three MoCs are FSM, SDF, and communicating sequential processes. SystemC-H imposes stylistic guidelines to increase the modeling fidelity [7] of SystemC. Details with code examples and descriptions are given in [9]. Although SystemC-H supported heterogeneity, it lacked heterogeneous behavioral hierarchy, disallowing designers to hierarchically compose designs with varying MoCs. In addition, they required significant alteration to the original SystemC scheduler, which we attempt to improve on. This paper shows that we can represent MoCs and their interactions as simple libraries which can be linked with SystemC models without altering the standard reference implementation.

C. Metropolis

Another University of California, Berkeley, group works on a project called Metropolis, whose purpose is again directed toward the design, verification, and synthesis of embedded software. However, their approach is different than Ptolemy II's. Metropolis has a notion of a metamodel as a set of abstract classes that can be derived or instantiated to model various communication and computation semantics. The basic modeling elements in Metropolis are processes, ports, media, quantity manager, and state media. Processes are atomic elements describing computations in its own thread of execution that communicate through ports. The ports are interfaced using media and the quantity manager enforces constraints on whether the process should be scheduled for execution or not. The quantity manager communicates through a special medium called state media. The communication elements such as the media are responsible for yielding a platform-based design that implements a particular MoC. They use a refinement-based methodology whereby starting from the most abstract model, they refine to an implementation platform. If heterogeneity is required, the user must provide a new media which implements the semantics for that MoC. A refinement of the model uses this new media for communication between the processes. Unfortunately, the refinement-based methodology disallows heterogeneous behavioral hierarchy because the heterogeneous components require a communication media between the two to transfer tokens causing them to be at the same level of hierarchy.

D. Statecharts and Starcharts

One of the most well-known contributions to concurrent hierarchical FSMs is Harel's Statecharts semantics [16]. Statecharts has the notion of OR and AND semantics. The OR states refer to states that have a hierarchically embedded FSM within it, which represent one of its hierarchically embedded states. The AND states contain multiple hierarchical embedded FSMs within a state that execute orthogonally. There are numerous variants of the Statecharts semantics due to the lack of strict execution definition during Harel's initial presentation, thus further coupling various concurrency models with Statecharts such as codesign FSMs. The execution definition describes the manner in which the transitions are enabled and the states are traversed, along with firing of the hierarchical states. For this reason, the Ptolemy II group proposed the Starcharts, whose semantics separate the concurrence MoC from the FSM semantics [15]. Their main contribution regarding HFSMs is in providing an execution definition for hierarchical FSMs void of any concurrence models. The semantic separation between Starcharts execution semantics from any other concurrence model motivates us to base our FSM MoC extension for SystemC using the Starcharts semantics. However, there are some fundamental qualities of Harel's Statecharts such as AND concurrence that we incorporate in our realization of the FSM extension, thus extending the Starcharts semantics [8].

V. Simulation Semantics for Heterogeneous Behavioral Hierarchy

Introducing multi-MoC support for hierarchy is a more difficult problem than either just a hierarchical FSM or SDF MoC by itself. Heterogeneous behavioral hierarchy requires well-defined semantics across the MoCs for the implementation to have an elegant solution for adding multi-MoC support for hierarchy. Furthermore, each MoC must have a clean mechanism for interacting with other MoCs such as initializing, invoking iterations, etc. It must also have tidy mechanisms to return control back to the MoC that invoke iterations on other MoCs. To aid us in fulfilling these requirements, we introduce the idea of abstract semantics interface that all executable entities implement. The directors also implement this interface and we present a formal representation of the definitions and algorithms used for the hierarchical FSM and SDF MoCs. We present the basic definitions and follow that with the hierarchical FSM algorithms in Fig. 3. This formal representation describes the well-defined execution semantics for behavioral hierarchy. However, by redefining a few of the basic definitions, we enable heterogeneous behavioral hierarchy by incorporating the SDF MoC whose algorithms are shown in Fig. 2. We borrow the SDF semantics from SystemC-H and point the readers to [7], [9], [12] for detailed information on the semantics behind scheduling and firing of SDF models. Hence, we only present details of the HFSM MoC implementation and explain how we enable heterogeneous behavioral hierarchy with the addition of the hierarchical SDF MoC.

A. Abstract Semantics

Our implementation follows an abstract semantics similar to Ptolemy II's `prefire()`, `fire()`, `postfire()`. Our approach declares `prepare()`, `precondition()`, `execute()`, `postcondition()`, and `cleanup()` as shown in Listing 4. Each of these pure virtual member functions have a specific responsibility. The `prepare()` member function initializes state variables and the executable entity, respectively. One iteration is defined by one invocation of `precondition()`, `execute()`, and `postcondition()`, in that order. The `cleanup()`'s responsibility is in releasing allocated resources. The `prepare()` and `cleanup()` member functions are only invoked once during initialization and then termination of the executable entity. For

Simulation Algorithm 1: **sdf_director's prepare(...)** – Prepare SDF block objects in director.

Require: $S = \emptyset$, $\beta \neq \emptyset$
  $\forall\ \alpha \in \beta$ **do**
    prepare(α)
  $S = \Gamma(\beta)$
  return true

Simulation Algorithm 2: **sdf_director's precondition(...)** – Precondition for SDF director.

  return true

Simulation Algorithm 3: **sdf_director's execute(...)** – Execution of an SDF director.

Require: $S \neq \emptyset$
  $\forall\ \alpha \in S$ **do**
    if $\rho(\alpha)$ = true **then**
      $trigger(\alpha)$
    end if
    $execute(\alpha)$
  return true

Simulation Algorithm 4: **sdf_director's postcondition(...)** – Postcondition for SDF director.

  return true

Simulation Algorithm 5: **sdf_director's cleanup(...)** – Cleanup for SDF director.

  $clean\_up()$
  return true

Fig. 2. Simulation algorithm. Starting from a heterogeneous hierarchical SDF model.

example, the `prepare()` of the `sdf_model` is responsible for computing the schedule as shown in Fig. 2, which only executes once, after which iterations of the entity are performed. Note in Listing 4, the return types of each of the virtual members. These return values are important in determining when the next semantically correct virtual member is to be executed. For example, the `precondition()` must be followed by `execute()`, which only occurs if the `precondition()` returns a `true` value. Similarly, the other virtual members return Boolean values signifying the next virtual member that it can execute. Every executable entity in our implementation follows this approach

Listing 4. Executable entity interface

```
1 class abs_semantics
2 {
3 public :
4   virtual bool prepare() = 0;
5   virtual bool precondition() = 0;
6   virtual bool execute() = 0;
7   virtual bool postcondition() = 0;
8   virtual bool cleanup() = 0;
9   abs_semantics();
10  virtual ~abs_semantics();
11 }
```

It is important to understand the simplicity of this approach and the benefit achieved when introducing heterogeneous behavioral hierarchy. Take as an example where a hierarchical FSM model contains several refinements of SDF models. Initialization of this model requires stepping through every executable entity and invoking the `prepare()` member function. The abstract semantics promote this by enforcing each of the member functions to fulfill a particular responsibility. For instance, the `prepare()` member function is only used to initialize state variables and other items for the executable entity. This categorization via the abstract semantics helps in invoking objects from heterogeneous MoCs through each other's executable interface. This approach allows us to implement heterogeneous behavioral hierarchy elegantly.

B. Basic Definitions

The basic definitions constitute of defining the sets for FSM states, transitions, and SDF block objects. This is followed by defining functions to check if an object has a refinement, to distinguish the type of the object, and whether the run-to-complete property of an object is set. Then, we define a function to iterate through all refinements of an object. Please note that we redefine these basic definitions when discussing our implementation.

Definition 5.1: Let β be the set of SDF block objects. Let $S \subseteq \beta$ be the list of SDF block objects in their correct firing order. Let ST be the set of state objects. Let TR be the set of transition objects.

Definition 5.2: To distinguish the type of an element, we have

$$\tau(x) = \begin{cases} \text{SDF} & x \in \beta \\ \text{FSMST} & x \in \text{ST} \\ \text{FSMTR} & x \in \text{TR} \end{cases}.$$

A refinement is a slave subsystem embedded within a component. In the following definition, a refinement may be embedded within a state or a transition, which means a slave HFSM may exist in the states or in transitions.

Definition 5.3: Check if entity has refinement

$$\rho(x) = \begin{cases} \text{true}, & x \text{ has at least one refinement} \\ \text{false}, & x \text{ has no refinements} \end{cases}$$

where $x \in \{\beta \cup \text{ST} \cup \text{TR}\}$.

Definition 5.4: Check if object is set to run-to-complete.

$$\text{run\_to\_complete}(x) = \begin{cases} \text{true}, & \text{object } x \text{ is set to run-to-complete} \\ \text{false}, & \text{object } x \text{ is not set to run-to-complete} \end{cases}.$$

Definition 5.5—trigger: Iterate through all refinements for particular object. Let R be the set of refinements for particular object. Let get_all_refs(x) return all refinements for object x where $x \in \{\beta \cup \text{ST} \cup \text{TR}\}$. Let $iterate(x)$ perform an iteration on the refinement of object x.

trigger(x) = **Require**: $R = \emptyset$
  R = get_all_refs(x)
  $\forall\ r \in R$ **do**
    $iterate(r)$

Simulation Algorithm 6:
fsm_director's prepare(...) – Prepare FSM objects in model.
Require: $ST \neq \emptyset$, $TR \neq \emptyset$, $initST \neq \emptyset$, $finalST \in \{ST \cup \emptyset\}$, $currST = initST$
Require: $initST \neq finalST$
$\forall\ \alpha \in \{ST \cup TR\}$ **do**
prepare(α)
return true

Simulation Algorithm 7:
fsm_director's precondition(...) – Precondition for FSM model.
Require: $currST \neq \emptyset$
$exec\_entry\_actions(currST)$
return true

Simulation Algorithm 8:
fsm_director's execute(...) – Execution for FSM director.
Let $get\_enabled\_tr(x)$ return an enabled transition on object x, where $x \in \{ST \cup \emptyset\}$
Require: $TR \neq \emptyset$, $currST \neq \emptyset$
$\alpha = get\_enabled\_tr(currST)$
if $\pi(\alpha) = PREEMPTIVE$ & $\rho(\alpha) = true$ **then**
$trigger(\alpha)$
$execute\_choice\_actions(\alpha)$
return true
end if
$trigger(currST)$
if $\pi(\alpha) = NONPREEMPTIVE$ & $\rho(\alpha) = true$ **then**
$trigger(\alpha)$
$execute\_choice\_actions(\alpha)$
end if
return true

Simulation Algorithm 9:
fsm_director's postcondition(...) – Postcondition for FSM director.
Let $lastTR$ be the last enabled transition where $lastTR \in \{TR \cup \emptyset\}$

Require: $lastTR$, $currST \neq \emptyset$
$execute\_commit\_actions(lastTR)$
$exec\_exit\_actions(currST)$
$currST = next\_state(lastTR)$
if $\pi(lastTR) = RESET$ **then**
$st\_reset(curST)$
end if
return true

Simulation Algorithm 10:
fsm_director's cleanup(...) – Cleanup for FSM director.
$clean\_up()$
return true

Fig. 3. Simulation algorithm. Starting from a heterogeneous hierarchical FSM model.

Algorithm 1 Execution semantics for HFSM
Let currST be the current state
$\alpha =$ get_enabled_tr (currST)
if $\alpha \neq \emptyset$ **then**
if $\rho(\alpha) = true$ **then**
trigger(α)
end if
execute_choice_action(α)
end if
{Execute state refinements}
if ρ(currST) $= true$ **then**
trigger(curST)
end if
{Execute this before state change}
if $\alpha =$ get_last_enabled_tr() **then**
execute_commit_action(α) **then**
currST $=$ next_state(α)
if $\pi(\alpha) =$ RESET **then**
reset_state(currST)
end if
end if

C. Execution Semantics for Starcharts

We describe the Starcharts execution semantics using the algorithm presented in Algorithm 1. However, to provide an intuitive understanding of hierarchical FSMs, let us take the simple example shown in Fig. 1(a) and assume that the `_rmt_AND` state does not exist. Therefore, the state machine only allows for incrementing and decrementing the channels using only the TV. With this altered example, the master FSM consists of `tv_on` and `off_tv`. The `tv_on` state has a refinement with three states `last_chnl`, `up_tv_chn`, and `down_tv_chn`. Once the TV is turned on, the master FSM enters the `tv_on` state which prompts the iteration of its refinement setting the TV to the last viewed channel. Once the last viewed channel is displayed, the refinement yields control to the master FSM, which checks whether the `turn_off` transition is enabled. If it is not, then the master FSM remains in the `tv_on` state. Similarly, while being in `tv_on` state, the channels are changed using the TV then the refinement's FSM changes state.

Using Algorithm 1, the current state of the HFSM as currST, the get_enabled_tr() retrieves an enabled transition from currST and ensures that there is a maximum of one enabled transition from a state to satisfy the determinism property of Starcharts. If the returned transition has any refinements embedded within it checked using the $\rho()$ function, then it performs iterations on all the refinements and its embedded refinements via the trigger procedure. Once the iterations are completed, the choice action of the enabled transition is triggered. This is used when modeling using heterogeneous MoCs which require multiple iterations of the HFSM without it changing the state such as fix pointing when embedded in a CT domain. This is followed by the same checking of refinements on currST so that transitions as well as states may have refinements within them. Finally, before changing the state of the topmost FSM, the commit actions are triggered and state is changed. If the last enabled transition is marked as a RESET transition, then the destination state's refinement is set to its initial state configuration. Note that this algorithm simply gives the basic flavor of the semantics which we further refine in Figs. 2 and 3 using the abstract semantics.

D. Execution Semantics for Hierarchical FSMs

Continuing to use the definitions presented earlier, we provide few additional definitions that allow us to present our algorithm for the execution semantics of HFSMs. Our algorithm is a variant of the Starcharts in that we incorporate a useful run-to-complete property for all objects. This allows for an object to iterate until its termination point. For example, a state set with the run-to-complete property in an HFSM only returns control to the master FSM after successfully traversing the refinement's HFSM from its initial to final state.

1) Additional Definitions Specific for HFSMs: The rtc_iterate() function performs run-to-complete iterations on the refinements of the object x. In the current definition, it

invokes a function that is specific to the HFSM MoC, rtc_fsm_iterate().

Definition 5.6: Run-to-complete iterate for different HFSMs

$$\text{rtc\_iterate}(x) = \text{rtc\_fsm\_iterate}(x)$$
$$\text{where } x \in \{\text{ST} \cup \text{TR}\}.$$

The get_all_refs() returns all the refinements in an object because an object may contain multiple HFSM refinements.

Definition 5.7: Return refinements for particular object of different MoC

$$\text{get\_all\_refs}(x) = Y'$$
$$\text{if } \tau(x) = \text{FSMST}|\text{FSMTR}, \text{then } Y' \subseteq \{\text{ST} \cup \text{TR}\}$$
$$\text{where } x \in \{\text{ST} \cup \text{TR}\}.$$

The $\pi()$ function determines the different types of possible transitions. Lastly, we redefine the trigger() procedure to incorporate run-to-complete iterations into the execution semantics.

Definition 5.8: To distinguish the type of transition objects

$$\pi(x) = \begin{cases} \texttt{PREEMPTIVE}, & x \text{ is set to preemptive} \\ \texttt{NONPREEMPTIVE}, & x \text{ is set to nonpreemptive or default} \\ \texttt{RESET}, & x \text{ is set to reset} \\ \texttt{RUN}-\texttt{TO}-\texttt{COMPLETE}, & \text{run-to-complete}(x) = \text{true} \end{cases}$$
$$\text{where } x \in \text{TR}.$$

An enabled transition of type PREEMPTIVE skips execution of the current state's refinement. The RESET transition resets the refinement of the destination state. Transition of type NONPREEMPTIVE executes the refinement of the transition, followed by executing the current state's refinement. The RUN−TO−COMPLETE transition semantics allows the particular refinement to execute until the FSM reaches the final state, after which the current state of the FSM is changed.

Iterate through all refinements for particular object. Let R be the set of refinements for particular object. Let get_all_refs(x) return all refinements for object x where $x \in \{\beta \cup \text{ST} \cup \text{TR}\}$. Let $iterate(x)$ perform an iteration on the refinement of object x. Let rtc_iterate(x) perform run-to-complete iterations on the refinements of object x

trigger(x) =
  require $R = \emptyset$
    $R = \text{get\_all\_refs}(x)$
    $\forall\, r \in R$ **do**
      if run_to_complete(r) **then**
        rtc_iterate(r)
      else
        $iterate(r)$
      end if

2) Redefinition for Heterogeneous Behavioral Hierarchical FSMs and SDFs: Once again, using the definitions and algorithms presented so far, we redefine some of the functions in order to allow for heterogeneous behavioral hierarchy. Note that it is simple to add the SDF MoC to the HFSM MoC. We only have to redefine the functions responsible for iterating through the refinements and adding one for the computation of the executable schedules for SDFs.

Definition 5.10: Compute the executable schedule for an SDF graph

$$\Gamma(X) = X', \quad \text{where } X \subseteq \beta.$$

Definition 5.11: Run-to-complete iterate for different MoCs

$$\text{rtc\_iterate}(x) = \begin{cases} \text{rtc\_sdf\_iterate}(x) & \text{if } \tau(x) = \text{SDF} \\ \text{rtc\_fsm\_iterate}(x) & \text{if } \tau(x) = \text{FSMST}|\text{FSMTR} \\ & \text{where } x \in \{\beta \cup \text{ST} \cup \text{TR}\} \end{cases}.$$

Definition 5.12: Return refinements for particular object of different MoC

$$\text{get\_all\_refs}(x) = \begin{cases} X' & \text{if } \tau(x) = \text{SDF, where } X' \subseteq \beta \\ Y' & \text{if } \tau(x) = \text{FSMST}|\text{FSMTR} \\ & \text{where } Y' \subseteq \{\text{ST} \cup \text{TR}\} \\ & \text{where } x \in \{\beta \cup \text{ST} \cup \text{TR}\} \end{cases}.$$

The only changes incurred when adding the hierarchical SDF MoC to the execution semantics was adding an additional case in both the rtc_iterate() and get_all_refs() functions. The former invokes a function specific for iterating through the SDF block objects and the latter for retrieving the refinements given that the object in question is an SDF block. Redefining these functions allows us to keep the simulation algorithms shown for the FSM and SDF model entities in Figs. 2 and 3 unaltered. In particular, note the underlined invocations because these are responsible for giving control to a refinement's respective simulation kernel. For example, in Fig. 2, the call to trigger performs one iteration of every refinement and its refinements for that one SDF block by giving control to the respective simulation control, executing one iteration and then returning control back to α.

E. Exit/Return and Reset Semantics

It is important to note that models created using our extensions execute in a single process, thus making the exit semantics relatively simple. Suppose that we have a model where an SDF component is embedded in a DE component, then once the SDF component begins executing, the only way it returns control to the master MoC is after completing one iteration of the executable schedule. We see this from Fig. 2's execute() member function, which only returns after all SDF blocks in S complete executing. Hence, one iteration of the SDF model is defined as one execution of the precondition(), followed by execute() and lastly postcondition().

Similar to the SDF's definition of an iteration, the FSM MoC's iteration consists of executing the precondition(),

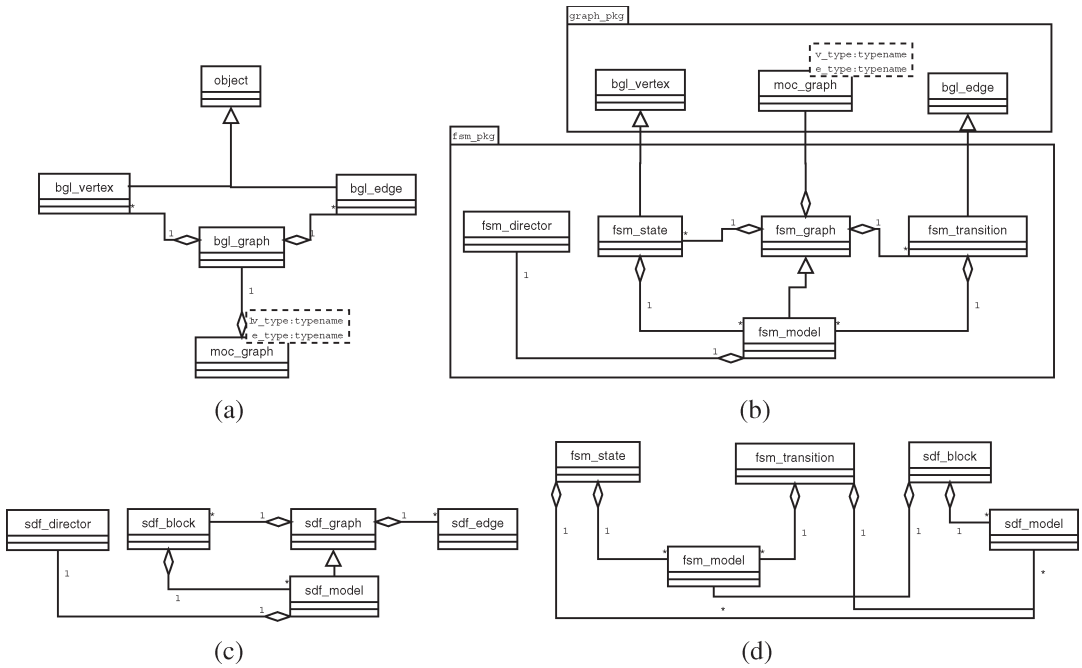


Fig. 4. Class diagrams. (a) BGL graph library. (b) FSM library. (c) SDF library. (d) Heterogeneous hierarchy.

`execute`(), and `postcondition`() as presented in Fig. 3. Once again, suppose a model has a refinement that follows the FSM MoC, then it returns control to the master kernel only after an iteration of the FSM model. These exit semantics show that the extended MoCs are noninterruptible during their iteration.

The reset semantics are applicable when the master is an FSM model, which may have slave refinements of either FSM or SDF MoCs. A transition object of type `RESET` in an FSM model invokes the `st_reset`() member function shown in Fig. 3's `postcondition`. This sets the master FSM's current state to the initial state and reinitializes every refinement in the FSM model by invoking the `prepare`() member function of the `fsm_director`. Hence, if there exists an FSM refinement in one of the states that reached its final state, then it is reset to its initial state. As for an SDF refinement in the master FSM, we do not require recomputing the executable schedule [7], [9] but simply invoke the `prepare`() member function as in Fig. 3. This will invoke `prepare`() on all the SDF blocks which will, in turn, prepare all the refinements in any of the SDF blocks.

VI. Implementing Heterogeneous Behavioral Hierarchy

We implement the hierarchical FSM and SDF algorithms in our extensions. However, our goal in making a clean and extensible implementation for hierarchical heterogeneity made it difficult to reuse all the implementation of the SDF and FSM MoCs from SystemC-H. First, the simulation semantics were tightly coupled with the DE scheduler, making it difficult to easily separate the two. Furthermore, the data structures used to represent the graphlike structure of SDF models was not generic. Evidently, most MoCs require a manner to represent the models. Therefore, the first requirement in implementing heterogeneous MoCs is to have a generic library to represent the models. We briefly describe our implementation of the graph library.

A. Graph Representation Using Boost Graph Library (BGL)

The first task in implementing any of these MoCs is their internal representation, which we represent using directed graphs. We implement our graph library using the BGL [21] and reuse this graph library for the FSM and SDF MoCs. For enabling hierarchy, it is crucial that the graph infrastructure supports hierarchical graphs. Fig. 4(a) shows the class diagram for the generic graph library.

1) Enabling Systemc With Hierarchical FSMs: Fig. 4(b) shows our representation of states and transitions in the `fsm_state` and `fsm_transition` classes, each inheriting from the `bgl_vertex` and `bgl_edge` classes. Likewise, the `fsm_graph` class inherits from the `moc_graph` class and the relationship between `fsm_state` and `fsm_transition` is of multiple containment. The `fsm_model` class inherits the `fsm_graph` class signifying that one instance of the `fsm_model` models one FSM. This is conveniently designed to allow FSM hierarchy within either states or transitions, noticeable by the relationship between the `fsm_state` and `fsm_transition` classes and the `fsm_model` class. The multiple containment relationship shows that there can be more than one instance of `fsm_model` embedded within either a state or transition. The role of the `fsm_director` is to implement the execution definitions for that particular MoC. In essence, it is the simulation kernel's responsibility for simulating that particular FSM. This is an important characteristic of our extensions. Earlier, we discussed preserving behavioral hierarchy during simulation and having separate simulation kernels executing their respective models does just that. We preserve the behaviors by making

each simulation kernel responsible for that refinement only, and likewise the state space abstraction is maintained.

B. Implementing the Execution Semantics

Fig. 3 shows that `prepare()` member function verifies that the user defined initial and final states are not the same. After this check verifies, `prepare()` is invoked on all executable entities, which includes states, transitions, and their refinements. All states and transitions are ready to execute, thus `precondition()` shows that there is no special processing required.

The `execute()` member function in Fig. 3 follows the semantics defined earlier in Section V-C with the exception that there are three different types of transitions and an additional property of the states and transitions. Note the three types of transitions enable specific functions of the refinements. They are preemptive, nonpreemptive, and reset. The function that checks this is $\pi()$. The additional property is a run-to-complete property for states and transitions.

There are two types of actions, commit and choice actions (terminology borrowed from the Ptolemy II project). The difference between choice and commit actions is that commit actions are only performed before the change of state for the FSM, whereas choice actions are performed whenever the corresponding transition is enabled. A possible scenario for using choice actions is embedding an FSM within a CT model, which requires the model to converge to a fixed point. During modeling HFSMs, we mainly use commit actions since we only want actions to be performed just before the change of states.

In Fig. 3, `postcondition()` performs the commit actions for the last enabled transition, changes the current state to the next state, and resets the refinement if the transition was of the `RESET` type. The `cleanup()` cleans up resources by resetting the values.

1) Enabling SystemC With Hierarchical SDFs: The implementation of the SDF MoC once again takes advantage of the graph library and follows a similar approach as shown for hierarchical FSMs. We restructure the source code from SystemC-H's implementation of the SDF MoC and reuse the scheduling algorithms to incorporate the graph library we implement. Fig. 4(c) shows the class diagram for the SDF library. Once again, we map the SDF MoC's structural representation to a graph by identifying the SDF blocks as vertices and the edges that connect the SDF blocks as edges from the graph library. Thus, the `sdf_block` inherits the `bgl_vertex` class and the `sdf_edge` inherits the `bgl_edge` class. The `sdf_block` class represents the computation block, which is connected by `sdf_edge`s. This connected representation of the SDF is represented by an `sdf_graph` class that realizes the graph structure for the SDF, but in order to allow for hierarchy, we derive the `sdf_model` class. This `sdf_model` class has a multiple containment relationship with the `sdf_block` class to show that hierarchical SDF models can be embedded within SDF blocks. However, this is not the case with `sdf_edge`s because in SDF they represent first-in first-outs and all computation is modeled in the SDF blocks. This organization of the SDF MoC prepares itself for heterogeneous hierarchy very well, because if an executable entity of any MoC was to allow embedding of hierarchical SDFs, then it exhibits a containment relationship with the `sdf_model` class. That would allow that particular executable entity to have refinements of the SDF MoC.

2) Enabling SystemC With Hierarchical SDFs and FSMs: Enabling heterogeneous hierarchy between SDF and FSM models is shown using snippets of the class diagram in Fig. 4(d). The important relationships to notice are again of multiple containment between the `fsm_state` and `fsm_transition` and that of `sdf_model`, as well as the relationship between `sdf_block` and `fsm_model`. This relationship suggests that an FSM state or transition can contain hierarchical SDF models within itself. Likewise, an SDF block can have hierarchical FSMs embedded within itself. The corresponding execution definitions and semantics are added in the MoC's respective directors. We understand that using these techniques we can integrate most other MoCs.

3) Integrating Extensions With SystemC: Our implementation of the extensions do not incur any changes to SystemC's reference implementation and we employ the same mechanism for integrating it with the DE scheduler as proposed in [9]. The intuition is to treat SystemC's scheduler as the master kernel that executes the respective extended kernels. The basic idea is to wrap a model constructed using the extensions in any SystemC-based process and invoking the `trigger()` of that MoC's model. We understand that there are several limitations when integrating using this approach. The foremost limitation is that if a refinement using the extension MoC contains a DE component in it, then there is an overlap of the hierarchy in that the DE components will be visible to the master scheduler due to the flattening of the hierarchy. Another difficulty is that SystemC follows a singleton design pattern making it impossible without alterations to preserve behavioral hierarchy for simulation. However, we reserve these detailed analysis for our future work and present our extensions as libraries. The FSM and SDF extensions are linked into one library such that additional extensions can also be easily integrated into the hierarchy enabled extensions. The only dependence our library has is that of the BGL available for free at [21]. Once BGL is installed, the extension's source can be compiled with any C++ compiler and linked to existing SystemC code by simply linking with the `libsch.a` library.

VII. Modeling Guidelines for the SDF Extension

We provide basic modeling guidelines that use snippets of code from the PIP example. In doing so, we only familiarize the readers with the essential macros required for using the extensions. The full source for the PIP example is available in [22]. The PIP is used for drawing polygons on the screen in a raster-based display system. More detail on its functionality is provided in Section VIII. An extensive comparison between using the extensions and SystemC's reference implementation is discussed in [10]. We have detailed the modeling guidelines for the hierarchical FSM in [8] with additional changes in [22]. Hence, in this section, we only present the SDF modeling guidelines. We also direct the readers to [22] for the source code for the PIP that employs several HFSMs.

Fig. 5. Component that verifies coordinate bounds.

A. Construction of SDF Model

An SDF model consists of SDF blocks, the SDF graph that shows the way in which these SDF blocks are connected to each other, and the production and consumption rates. As an example, we describe a component of the PIP that verifies the bounds of the coordinates specified for shading. It is a three-block SDF example as shown in Fig. 5. The first block specifies the coordinates, the second verifies that they are within the bounds, and the third inserts it into structures allowing for further processing

Listing 5. SDF query block using extensions

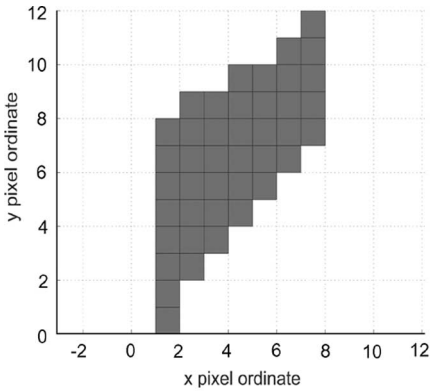
```
1 SCH_SDF_MODULE()(query)
2 {
3   public:
4     behavior_interface ⟨int⟩ x_out;
5     behavior_interface ⟨int⟩ y_out;
6     int in_cnt;
7     SCH_SDF_CTOR()
8     { in_cnt = 0; };
9
10     SCH_SDF_BLOCKACTION()
11       { // implementation code }
12   private:
13     int x, y;
14 }
```

Listing 5 shows the query SDF block's implementation. Note that we use the `SCH_SDF_MODULE()` to describe an SDF block. The behavioral interfaces in [Listing 5, Lines 4 and 5] act as interfaces to other SDF blocks. We require users to employ these interfaces, which are then connected using `behavior_tunnel`-type channels when constructing the SDF graph. The constructor in [Listing 5, Line 7] initializes a data member and internally informs the kernel of the instance name of the SDF module. Then, the `SCH_SDF_ACTION()` MACRO contains the actual computation of the SDF block. Similarly, the remainder of the blocks `verify` and `insert` are modeled using the same MACROs and similar data transfer mechanisms

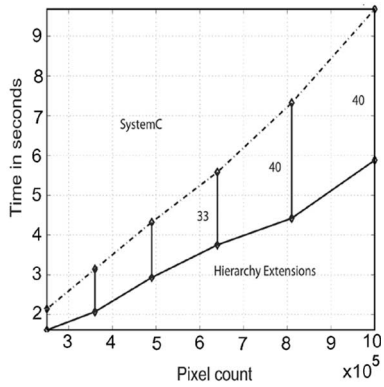
Listing 6. Constructing the SDF graph

```
1 SCH_SDF_MODULE( user_input )
2 {
3   public
4     query * first;
5     verify_input * second;
6     insert_vertices * third;
7     sdf_edge * first_second;
8     sdf_edge * second_third;
9     sdf_edge * third_first;
10     behavior_tunnel ⟨int⟩ query_verify_x;
11     // more tunnels
12     behavior_tunnel ⟨int⟩ verify_insert_y;
13
14     SCH_SDF_MODEL_CTOR(user_input)
15       {
16       first = new query ("Query");
17       second = new verify_input("VerifyInput");
18       third = new insert_vertices("InsertVertices");
19
20         first_second = new sdf_edge
           ("query_to_verify");
21         second_third = new sdf_edge
           ("verify_to_insert");
22         third_first = new sdf_edge ("loopback");
23
24         first->x_out (query_verify_x);
25         // more implementation
26         third->y_in (verify_insert_y);
27
28         first_second->set_production_rate (1);
29         // more implementation
30         third_first->set_consumption_rate (1);
31         third_first->set_delay(1);
32         insert_sdf_block (first);
33         // insert remaining blocks
34         insert_sdf_edge
           (first, second, first_second);
35         // insert remaining edges
36       }
37 }
```

The `SCH_SDF_MODEL()` MACRO declares the SDF model in which the SDF graph is constructed. The first step in defining the SDF model is assigning pointers to the SDF blocks that are to be instantiated, shown in [Listing 6, Lines 4–6]. These pointers are later initialized in the constructor of this SDF model in [Listing 6, Lines 16–18]. We create SDF edges that connect the blocks together by declaring the pointers in [Listing 6, Lines 7–9] and later instantiated in the constructor in [Listing 6, Lines 20–22]. We also define the tunnels using `behavior_tunnel`-typed objects that connect to the interfaces defined in the SDF blocks using `behavior_interface`. The instantiations of the tunnels are shown in [Listing 6, Lines 10–12]. The user must specify the production and consumption rates on each of these edges using the `set_production_rate()` and `set_consumption_rate()` shown in [Listing 6, Lines 28–30]. For this model, we simply use single rates. Notice [Listing 6, Line 31] uses the `set_delay()` member function, which is responsible for notifying the kernel of the initial tokens on that particular edge when there are cyclical SDF graphs. More details regarding the scheduling of SDF graphs are discussed in [7] and [9]. After the rates are set on the SDF edges, the blocks and their connection with the edges are added into the model. We employ the `insert_sdf_block()` member function to insert the SDF blocks into the model and then the `insert_sdf_edge()` to connect the SDF blocks and insert this information into the model. The arguments passed into the `insert_sdf_edge()` member function are as follows: the block from which the edge



(a)



(c)

Pixels	SC(s)	SCH(s)	%Imp.
500^2	2.14	1.61	24
600^2	3.14	2.07	34
700^2	4.32	2.93	32
800^2	5.58	3.75	33
900^2	7.32	4.42	39
1000^2	9.68	5.88	40

(b)

Fig. 6. PIP results. (a) In-fill processor output. (b) Table showing simulation times. (c) Simulation results for in-fill processor.

is coming from, the block to which the edge is going to, and the edge that is connecting the two.

B. Tunnels and Interfaces for Extended MoCs

We implement templated classes `behavior_interface` and `behavior_tunnel` for data transfer for the extended MoCs. We do not employ any of the SystemC channels primarily because they employ the DE semantics. This also means that we cannot mix SystemC channels with our tunnels and interfaces. If data needs to be passed from one to another, then explicit reads from one and writes to the other must be performed. Therefore, connecting SystemC channels such as `sc_fifos` directly to an interface used in an SDF model is not possible at the moment. However, our tunnels and interfaces can use any SystemC data-type and they also allow for hierarchical binding of interfaces such that the same tunnel may be seen by hierarchical models.

VIII. Examples

A. Polygon In-Fill Processor (PIP)

This example shown in Fig. 7 implements a variation of the PIP from [23], which is commonly used for drawing polygons on the screen in a raster-based display system. The PIP is a good example of a heterogeneous design due to the various control machine and dataflow components it contains within itself. An example output of the in-fill processor plotted using Matlab is shown in Fig. 6(a).

The PIP is a heterogeneous system composed of five behavioral components. Three of the components follow the FSM MoC and the other two adhere to the SDF MoC. The `master.FSM` is the master controller which constitutes of the `init`, `inpt`, and `hrtr` states. The transitions between these states are guard/action pairs, which we annotate by a letter from the alphabet such as $\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \ldots$. States `inpt` and `hrtr` are both refined with SDF subsystems. Transition `d` has an FSM refinement embedded as a `run-to-complete`. Similarly, `d.FSM` has three states including the `finl` state, to indicate that one iteration of the FSM is complete. State `comp` is refined with a `run-to-complete` FSM that computes Bresenham's line algorithm. Note that Fig. 7 describes our implementation at a higher level of abstraction without specifics on the transition guards, production and consumption rates, or the `behavior_interfaces` and `behavior_tunnels` used to transport data within and across the components.

The user input is modeled in the `inpt` state as an SDF refinement `input.SDF`. The line computation for the four vertices occurs in transition `d` with `d.FSM` responsible for computing lines for each pair of coordinates and `compute.FSM` computing Bresenham's line. The `hrtr` state performs the horizontal trace using an SDF refinement `hrtr.SDF`.

1) Results: We conducted simulation experiments for the in-fill processor example by implementing an optimized SystemC (`SC_METHOD`)-based version of the in-fill processor and compared the execution times between the two models. Fig. 6(c) shows the graph comparing the times taken to execute the models in both SystemC and our heterogeneous hierarchical enabled versions with varying pixel counts (the number of pixels shaded). Fig. 6(b) shows the tabulated values shown on the graph, where `SC` stands for SystemC and `SCH` is our heterogeneous behavioral hierarchy simulation extensions for SystemC. We see that our version outperforms the SystemC

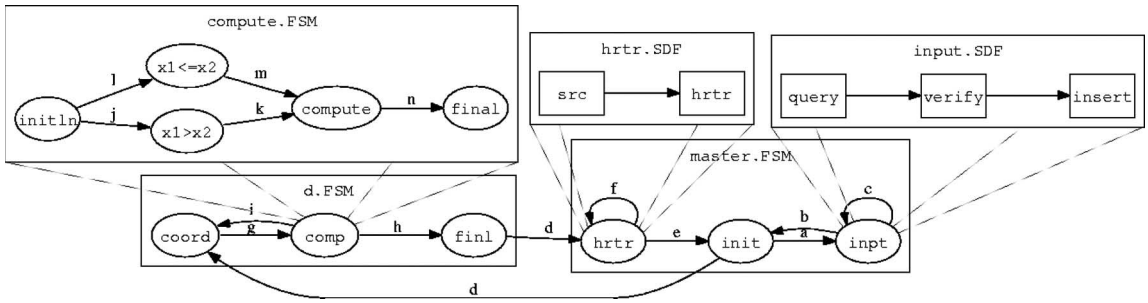


Fig. 7. Behavioral decomposition of in-fill processor.

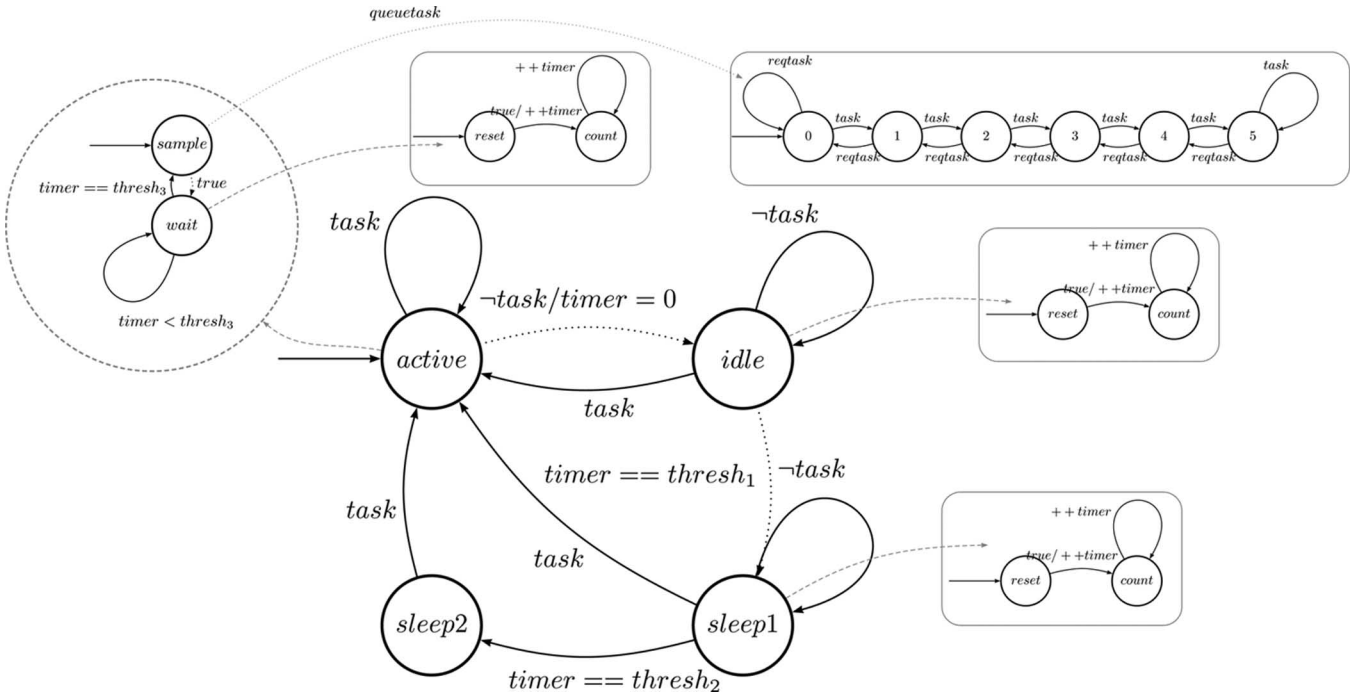


Fig. 8. Power model using hierarchical FSMs.

model by approximately 40% (the numbers present on the horizontal line between two of the data points show the percentage improvement). However, the performance improvement is not linear as can be seen by the results for the lower pixel count. This occurs because our extension for the SDF MoC requires static scheduling during initialization, which is computationally intense. On the other hand, SystemC's simulation kernel uses dynamic scheduling through the use of events, hence, not suffering from this static scheduling overhead. This overhead of scheduling outweighs the actual computation of the components for a low pixel count. However, when we increase the pixel count, we see that SystemC's execution time deteriorates whereas our extension's simulation time improves as shown in Fig. 6(c).

B. Abstract Power State Model

We present an example of a power model in Fig. 8. The model describes the topmost FSM containing the states `active`, `idle`, `sleep1`, and `sleep2` depicting one power saving state as `sleep1` and a deep power saving state as `sleep2`. The `active` state has an FSM slave subsystem that samples for arrival of tasks every thresh_3. For this, the `wait` state contains an FSM refinement of a timer. On a task arrival and when the sampler is in the sample state, the task id is queued onto a global queue with a maximum size of five ids. We limit the functionality of the queue to simply return the task id, and in the case of pushing a task when the queue is full, it simply discards the task. Similarly, if the queue is empty and a request for servicing the queue is received, the queue ignores the request. A timer is also included in the `idle` and `sleep1` states to allow transitions from `idle` to `sleep1` and then from `sleep1` to `sleep2` based on specific timing constraints. Transitions that are depicted with black dotted arrows signify reset transitions. Note that any transition that does not show a guard/action pair but only the guard, assumes that there is no action associated with the transition besides state change.

1) Results: We implement the power model using our HFSM SystemC extension as well as using the reference implementation of SystemC and compare simulation speeds for the two models. The results are shown in Table I.

TABLE I
SIMULATION SPEEDS FOR POWER MODEL

Samples	SystemC (s)	HFSM (s)
10000	51.83	50.84
20000	103.46	101.9
30000	159.21	153.18
40000	208.47	205.28
50000	260.98	255.14

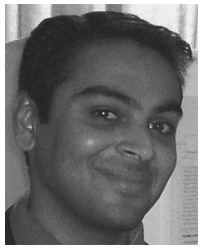
The two models traverse the FSMs in approximately the same order and the HFSM model shows a 2% improvement over the SystemC model. Since there are no optimizations such as static scheduling that can be performed on an FSM model, we find no significant simulation speedup as acceptable because we have increased modeling fidelity. We attribute this slight increase in simulation efficiency to the model using the extensions not requiring the use of SystemC processes. However, the advantages of heterogeneous behavioral hierarchy is evident from the PIP example.

IX. CONCLUSION AND FUTURE WORK

In this paper, we enable our multi-MoC extensions with an important characteristic of heterogeneous behavioral hierarchy with the addition of the hierarchical SDF MoC alongside the hierarchical FSM MoC. The issue of adding multi-MoC support for hierarchy is nontrivial as it requires defining and understanding semantics across MoCs. We present our extensions for SystemC with heterogeneous hierarchy and display class diagrams and algorithms to give a better idea of our approach. We present a hierarchical FSM example of an abstract power state model and a heterogeneous hierarchy example of the PIP. The former does not show a significant increase in simulation speed but neither does it show any degradation, but the heterogeneous behavioral hierarchy example shows an approximately 40% improvement over its counterpart. We use the in-fill processor example to show a significant increase in simulation performance due to the preservation of hierarchy during simulation. However, the formalization of the semantics is a subject for future work.

REFERENCES

[1] SPECC, *SpecC*. [Online]. Available: http://www.ics.uci.edu/~specc/
[2] SystemVerilog, *System Verilog*. [Online]. Available: http://www.systemverilog.org/
[3] OSCI, *SystemC*. [Online]. Available: http://www.systemc.org
[4] Ptolemy Group, *Ptolemy II Website*. [Online]. Available: http://ptolemy.eecs.berkeley.edu/
[5] Metropolis Group, *Metropolis: Design Environment for Heterogeneous Systems*. [Online]. Available: http://embedded.eecs.berkeley.edu/metropolis/
[6] H. D. Patel. (2003, Dec.). "HEMLOCK: HEterogeneous ModeL Of Computation Kernel for SystemC," M.S. thesis, Virginia Polytechnic Inst. State Univ., Blacksburg, VA, [Online]. Available: http://fermat.ece.vt.edu
[7] 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. Comput.-Aided Design Integr. Syst.*, vol. 24, no. 8, pp. 1261–1271, Aug. 2005.
[8] ——, "Towards behavioral hierarchy extensions for SystemC," in *Proc. FDL*, 2005.
[9] ——, *SystemC Kernel Extensions for Heterogeneous System Modeling*. Norwell, MA: Kluwer, 2004.
[10] ——, "Deep vs. shallow, kernel vs. language——What is better for heterogeneous modeling in SystemC?" FERMAT Lab Virginia Tech., Blacksburg, VA, Tech. Rep., 2006-01.
[11] ——, "Towards a heterogeneous simulation kernel for system level models: A SystemC kernel for synchronous data flow models," in *Proc. Great Lakes Symp. VLSI*, 2004, pp. 241–242.
[12] FERMAT, *SystemC Kernel Extensions for Heterogeneous System Modeling: A Framework for Multi-MoC Modeling & Simulation*. [Online]. Available: http://fermat.ece.vt.edu/systemc-h/
[13] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Comput.*, vol. C-36, no. 1, pp. 24–35, Jan. 1987.
[14] S. Bhattacharyya, P. Murthy, and E. Lee, *Software Synthesis from Dataflow Graphs*. Norwell, MA: Kluwer, 1996.
[15] A. Girault, B. Lee, and E. A. Lee, "Hierarchical finite state machines with multiple concurrency models," in *Proc. IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, 1999, vol. 18, pp. 742–760.
[16] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. Comput. Program*, vol. 8, no. 3, pp. 231–274, Jun 1987.
[17] VERILOG, *Verilog*. [Online]. Available: http://www.verilog.com/
[18] VHDL, *VHDL*. [Online]. Available: http://www.vhdl.org/
[19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*. Reading, MA: Addison-Wesley, 1995.
[20] H. D. Patel and S. K. Shukla, "Heterogeneous behavioral hierarchy for system level designs," in *Proc. Des. Autom. Test Eur.*, 2005, pp. 1–6.
[21] Boost, *The Boost Graph Library*. [Online]. Available: http://www.boost.org/libs/graph/doc/index.html
[22] H. D. Patel, S. K. Shukla, *Heterogeneous and Behavioral Hierarchy Extensions for SystemC*. [Online]. Available: http://fermat.ece.vt.edu/tools/hierarchy/
[23] R. A. Bergamaschi, "The development of a high-level synthesis system for Concurrent VLSI systems," Ph.D. dissertation, Univ. Southampton, Southampton, U.K., 1989.



Hiren D. Patel (S'04) received the B.S. and M.S. degrees, both from Virginia Polytech and State University, Blacksburg, in 2001 and 2003, respectively, where he is currently working toward the Ph.D. degree.

His research interests include system level design methodologies and frameworks and models of computation/Multi-MoC modeling. His recent work involves introducing heterogeneous behavioral hierarchy in SystemC and service-oriented architectures for validating system level designs. He has published around 17 papers as journal and conference proceedings and has coauthored a book titled SystemC Kernel Extensions for Heterogeneous Modeling.

Sandeep K. Shukla (M'99–SM'02) is an Assistant Professor of computer engineering at Virginia Polytech and State University, Blacksburg. He is also a Founder and Deputy Director of the center for embedded systems for critical applications, and Director of his research lab FERMAT. He has published more than 80 papers in journals, books, and conference proceedings. He recently coauthored *SystemC Kernel Extensions for Heterogeneous Modeling*, and coedited *Nano, Quantum and Molecular Computing: Implications to High Level Design and Validation*, and *Formal Methods and Models for System Design: A System Level Perspective* (Kluwer).

Dr. Shukla was also recently awarded the Presidential Early Career Awards for Scientists and Engineers (PECASE) award for his research in design automation for embedded systems design, which in particular focuses on system level design languages, formal methods, formal specification languages, probabilistic modeling and model checking, dynamic power management, application of stochastic models and model analysis tools for fault-tolerant system design, and reliability measurement of fault-tolerant systems. He was elected as a College of Engineering Faculty Fellow at Virginia Tech. He also chaired a number of international conferences and workshops, edited a number of special issues for various journals, and is on the editorial board of IEEE DESIGN AND TEST.

Reinaldo A. Bergamaschi (S'86–M'88–SM'96–F'05) received the B.S. degree in electronics engineering from Aeronautics Institute of Technology, Brazil, in 1982, the M.S. degree in electronics engineering from Philips International Institute, Eindhoven, The Netherlands, in 1984, and the Ph.D. degree in electronics and computer science from the University of Southampton, Southampton, U.K., in 1989.

Since 1989, he has been with the IBM Thomas J. Watson Research Center, Yorktown Heights, NY. He has lead research and development projects on a variety of design automation areas including high-level synthesis, register-transfer language signoff techniques, early estimation of area, timing and power, systems-on-chip design automation and system-level modeling. He has published over 40 technical papers and served in conference committees for all major conferences in the design automation field.

Dr. Bergamaschi was the General Chair for the 2006 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISS).