Heterogeneous Behavioral Hierarchy for System Level Designs

Hiren D. Patel
Virginia Polytechnic and State University
hiren@vt.edu

Sandeep K. Shukla
Virginia Polytechnic and State University
shukla@vt.edu

Reinaldo A. Bergamaschi
IBM Research T.J. Watson
berga@us.ibm.com

ABSTRACT

Enhancing productivity for designing complex embedded systems requires system level design methodology and language support for capturing complex design in high level models. 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 for system level abstractions, several essential ingredients are missing from these. We consider (i) explicit support for multiple models of computation (MoCs) or heterogeneity; (ii) the ability to build complex behaviors by hierarchically composing simpler behaviors; and (iii) hierarchical composition of behaviors that belong to distinct models of computation, as essential for successful SLDLs. These render an SLDL with modeling fidelity that exploits both heterogeneity and hierarchy and allows for simpler modeling and efficient simulation. One important requirement for such an SLDL should be that the simulation semantics be also compositional, and hence no flattening of hierarchically composed behaviors be needed for simulation. In this paper we show how we designed SystemC extensions to provide facilities for heterogeneous behavioral hierarchy, compositional simulation semantics, and implemented a simulation kernel which we show experimentally as up to 50% more efficient than standard SystemC simulation.

1. INTRODUCTION

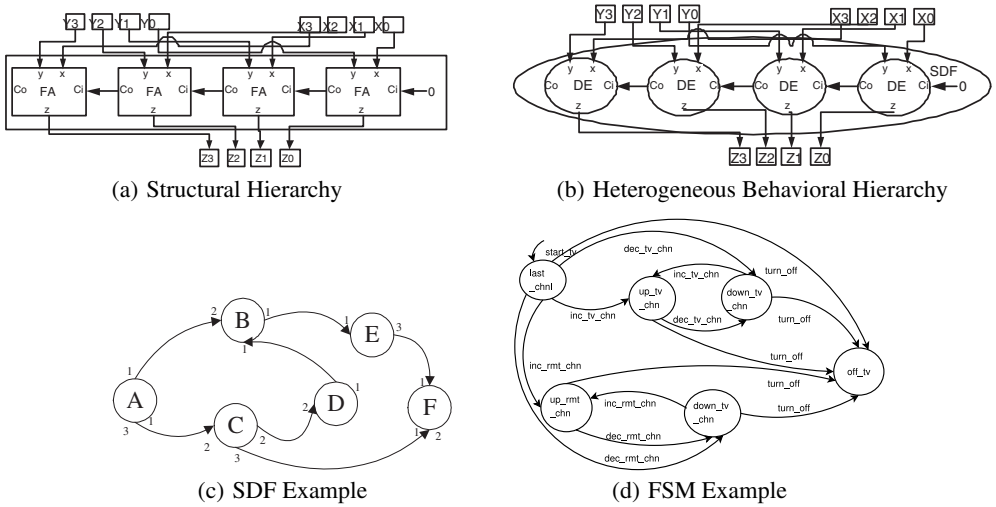
Various system level design languages (SLDLs) have been proposed for the specification and modeling of complex and heterogeneous hardware and software systems. SLDLs such as SpecC [14], SystemVerilog [15] and SystemC [9] provide system-level abstractions for hardware modeling, and to a lesser extent for software modeling. Environments like Ptolemy II [13] and Metropolis [5] address either control-oriented embedded software design and/or platform-based system design. These SLDLs have limitations, which restrict their widespread use in industry for modeling heterogeneous hardware and software systems. These limitations 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 in our view 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 (similar to VHDL and Verilog). Although other MoC semantics can be mimicked by DE, it complicates the modeling of fully heterogeneous systems (with different models of computation). Also, this imposes a significant overhead in code size and simulation speed, if such designs are modeled using the current SystemC implementation. As a result, adding the features we deem as necessary, and elaborate further in the paper, becomes challenging without adding simulation semantics distinct from the DE semantics, on top of adding abstraction mechanisms.

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

This third feature allows for simpler 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, and larger models are built via hierarchical composition. Furthermore, the simulation semantics for such SLDLs should be also compositional so that *flattening* of hierarchically composed behaviors is not needed for simulation. This paper presents methods and implementation of the above characteristics as extensions to the SystemC kernel, to allow it to natively model heterogeneous behavioral hierarchy and at the same time achieve simulation speeds up to 50% faster than standard SystemC simulation of the same design. This work significantly extends our previous work on heterogeneous simulation kernel extensions for SystemC.

The capability for heterogeneity in any SLDL allows decomposition of designs based on their inherent behaviors of the components, which we term *behavioral decomposition*. This allows the design to be realized as communicating sub-components which are expressed and governed by particular MoCs. *Behavioral hierarchy* [12] is the second extension of SystemC that we have created. We term behavioral hierarchy as the ability to build a design by composing smaller behaviors hierarchically. Large system models comprising of small behavioral sub-components may consist of smaller components embedded within the sub-components. Our extensions to SystemC support behavioral hierarchy encouraging designers to seamlessly compose behaviors hierarchically without having to worry about the semantics that define the composition. This is because we define compositional execution semantics for hierarchical compositions [12] which is 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, where an SoC may be modeled with a

3-9810801-0-6/DATE06 © 2006 EDAA



(a) Structural Hierarchy

(b) Heterogeneous Behavioral Hierarchy

(c) SDF Example

(d) FSM Example

Figure 1: Examples of SDF, FSM, structural and heterogeneous behavioral hierarchy

hierarchical controller using the hierarchical finite state machine (FSM) MoC, a digital signal processing (DSP) algorithm 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 sub-components 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 a sub-component) 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 as shown in Figure 1(a). 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. In Figure 1(b), we show the same adder design in behavioral form. Since this is too simple an example, it turns out that the structural and behavioral decomposition are isomorphic, but that may not always be the case. In the behavioral representation, a four-bit adder could be modeled as an SDF with four components, and each component expresses the equation representation of one-bit adder behavior, which may be simulated using discrete event (DE) semantics.

Current SystemC allows for structural hierarchy, but lacks support for behavioral hierarchy. It provides structural hierarchy through C++ composition technique, 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 a DE semantics. Moreover, during simulation, all these structurally embedded processes are treated at the same level thereby *flattening* any implied behavioral 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 both benefits during modeling and simulation. Behavioral decomposition of the design provides encapsulation and abstractions during modeling, whereas for simulation, the proposed simulation kernel extensions only simulate one level of hierarchy at any time. Thus preserving the state space abstraction along with the 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 will execute the sub-system. We contend that behavioral hierarchy is a key in not only increasing the modeling fidelity [11], but also aiding in simulation efficiency. Also note that our behavioral hierarchy is independent of structural hierarchy, and in our design, at the different levels of hierarchy, distinct MoCs may govern the behaviors.

In particular, in this paper we present our multi-MoC hierarchical SystemC extensions for the FSM and SDF MoCs, which are interoperable with the SystemC discrete event (DE) simulation kernel to support heterogeneous behavioral hierarchy. We demonstrate the modeling paradigm with a *polygon in-fill processor* (PIP) example. This nontrivial example shows that behavioral decomposition is a good means of dissecting complex designs into small behaviors and then composing the overall model hierarchically. However, due to page number restrictions, we do not provide an *ease-of-modeling* comparison for the PIP between the model constructed using our extensions versus SystemC. We simply state that the decomposed behaviors are naturally mapped onto the extensions and also that simulation efficiency is achieved. Hence, aside from the advantages of system level abstraction and well defined simulation semantics, we also demonstrate advantages in simulation efficiency. We report significant simulation improvement over an analogous implementation of the PIP in standard SystemC.

1.1 Main contributions

The main contributions of this paper are: (1) rendering current industry standard SystemC with the ability of *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 on an interesting example of a graphics in-fill processor how to model with the extended SystemC, and do simulation 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 50% simulation efficiency enhancement with our first-cut implementation of the new simulation kernel.

1.2 Organization

The remainder of the paper begins discussing some related work followed by the necessary background in Sections 2 and 3 respectively. We continue with our introduction of heterogeneous behavioral hierarchy by describing the SDF extension added to the FSM extension for SystemC. Section 5 discusses a comprehensive example of an in-fill processor and Section 6 presents simulation results compared to the reference implementation of SystemC. We finally conclude in Section 7.

2. 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 [13], Metropolis [5], YAPI [4] and SystemC-H [11]. Ptolemy II is one of the first promoters of heterogeneous and hierarchical embedded system designs. Even though Ptolemy II is the only design environment that supports heterogeneous behavioral hierarchy, it has its own limitations. Firstly, Ptolemy II is geared towards the embedded software and software synthesis community and not targeted towards hardware and SoC designers. Therefore, certain useful semantics that may be considered essential for hardware modeling may not be easily expressible in Ptolemy II [12]. From one of the discussions with the Ptolemy developers, we understand that the Starchart 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 concurrency 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. Secondly, hardware designers often need a single SLDL or framework to refine their models from a high-level description to generally an RTL abstraction. This would not be possible in a framework such as Ptolemy II. Lastly, there is a large IP base built using C/C++ that is easier to integrate with an SLDL built using C/C++ opposed to Ptolemy II. Metropolis is another project that allows platform based designs that follow particular MoCs. However, heterogeneous behavioral hierarchy is not possible in Metropolis because in their heterogeneous components require a communication media between the two to transfer tokens causing them to be at the same level of hierarchy. YAPI is a C++ run-time library specific for signal processing applications, thus already limiting its heterogeneity. SystemC-H supports heterogeneity but it lacks behavioral hierarchy, not allowing designers to hierarchically compose designs with varying MoCs.

Before implementing our hierarchical FSM (HFSM) extension in [12], we studied several semantics for the FSM MoC. Of those, one of the most well known contributions to concurrent hierarchical FSMs is Harel's Statecharts semantics [6]. Statecharts has the notion of OR and AND semantics. The OR states refer to states that have a hierarchically embedded FSM within themselves, which represent one of its hierarchically embedded states. The AND states are states that 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 co-design 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 emphasizes on separating the concurrency MoCs from the actual FSM semantics by proposing the Starchart (*chart) semantics [1]. Their main contribution regarding HFSMs is in providing an execution definition for hierarchical FSMs void of any concurrency models. The semantic separation between *chart execution semantics from any other concurrency model motivates us to base our FSM MoC extension for SystemC using the *chart semantics. However, there are some fundamental qualities of Harel's Statechart such as AND concurrency that we incorporate in our realization of the FSM extension, thus extending the *chart semantics [12].

3. BACKGROUND

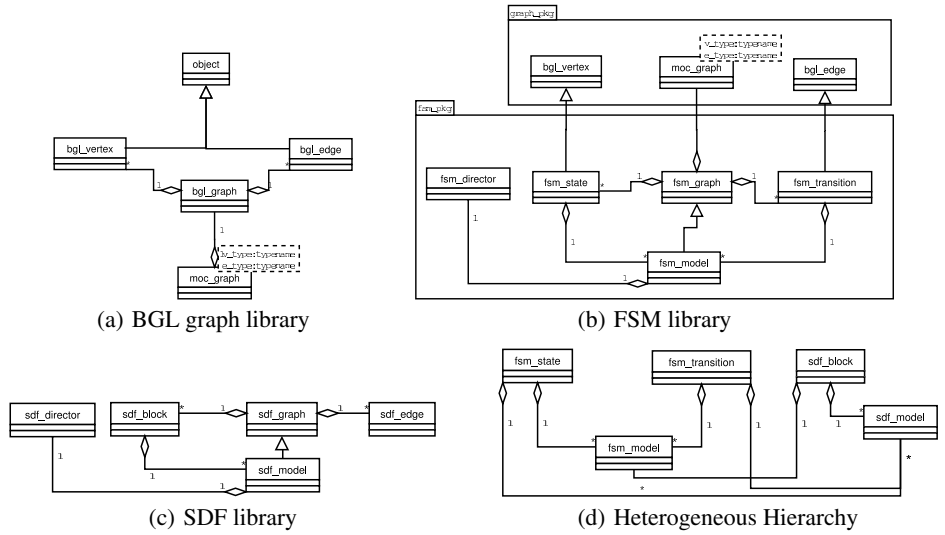
Models of Computation. An MoC describes the manner in which computation and communication occur in that particular MoC and when interacting with different MoCs [8, 11]. Examples of MoCs are DE, CT, SDF and FSM.

Synchronous Data Flow MoC. The synchronous data flow (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. The SDF MoC further restricts the data flow MoC by constraining the execution of the computation blocks only when there are sufficient data samples on the incoming data path arcs. This requires definition of production and consumption rates for the computation blocks on each respective data path. Thus, once a computation block has sufficient samples on its incoming data path (specified by the consumption rate for each arc), then it is fired and a specified number (production rate for each arc) of samples are expunged on the data path arcs. Figure 1(c) shows an abstract example of an SDF graph with the numbers on the head of the arcs as the production rates and the numbers at the tail as the consumption rates. An attractive quality of the SDF MoC is that an executable schedule can be computed during initialization [7, 11].

Finite State Machine MoC. Finite state machines (FSM) are usually 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. Figure 1(d) shows an example of a television button control FSM.

4. SIMULATION SEMANTICS FOR HETEROGENEOUS BEHAVIORAL HIERARCHY

Introducing multi-MoC support for hierarchy is a more difficult problem than simply hierarchical FSMs. Heterogeneous behavioral hierarchy requires an understanding of the semantics across the different MoCs. In addition, the implementation must also have an elegant solution for adding multi-MoC support for hierarchy. Our implementation follows an abstract syntax similar to Ptolemy II's `prefire`, `fire`, `postfire`. We follow a similar approach where we define `prepare`, `precondition`, `execute`, `postcondition` and `cleanup`. 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, respectively. For example, the `prepare` of the `sdf_model` responsible



(a) BGL graph library (b) FSM library

(c) SDF library (d) Heterogeneous Hierarchy

Figure 2: Class diagrams

{**sdf_model::prepare()**}
Initially: $S = \emptyset$ {List of firing order with SDF block objects}
Initially: $B \neq \emptyset$ {List with SDF block objects}
for all $bl \in B$ **do**
bl_prepare(bl) {Invoke prepare on every refinement in block}
end for
compute_schedule(S) {Store computed schedule in S}
return true

{**sdf_model::precondition()**}
return true

{**sdf_model::execute()**}
Initially: $S \neq \emptyset$
for all $bl \in S$ **do**
if sdf_ref(bl) **then**
iterate_sdf_ref(bl) {Invoke iterate on every SDF refinement}
end if
if fsm_ref(bl) **then**
iterate_fsm_ref(bl) {Invoke iterate on every FSM refinement}
end if
execute_block_actions(bl);
end for
return true

{**sdf_model::postcondition()**}
return true

{**sdf_model::cleanup()**}
clean_up()
return true

{**iterate_sdf_ref(E)**}
$R = get\_all\_sdf\_ref(E)$
for all $r \in R$ **do**
if run_to_complete(r) **then**
one_iterate(r)
end if
end for

Figure 3: Simulation algorithm: Starting from a Heterogeneous Hierarchical SDF model

for computing the schedule in Figure 3 only executes once, after which iterations of the entity are performed. Every executable entity in our implementation follows this approach. This approach allows us to implement heterogeneous hierarchy elegantly. For the SDF and FSM model entities, we show the algorithm employed in enabling heterogeneous hierarchy in Figure 3 and Figure 4. Particularly note the underlined operations because these are responsible for giving control to a refinement's respective simulation kernel. For example, in Figure 3 the call to `iterate_fsm_ref()` 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 `bl`.

This approach used in these algorithms is similar to Ptolemy II's in the manner in which control is transferred from one simulation kernel to another (referred to as directors in Ptolemy II). The one distinguishing factor is the addition of orthogonal FSMs and the run-to-complete FSMs.

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 [10]. 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 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 dependency our library has is that of the Boost Graph Library available for free at [3]. 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.

4.1 Implementing Heterogeneous Behavioral Hierarchy

Graph Representation. 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 Boost Graph Library (BGL) [3] and reuse this graph library for

```
{fsm_model::prepare()}
Initially: ST ≠ ∅ {List of state objects}
Initially: TR ≠ ∅ {List of transition objects}
Initially: initST ≠ ∅ {Initial state}
Initially: finalST {Final state}
Initially: currST = initST {Current state}
  if initST = finalST then
    return
  end if
  for all st ∈ ST, tr ∈ TR do
    st_prepare( st )
    tr_prepare( tr )
  end for
  return true

{fsm_model::precondition()}
return true
```

```
{fsm_model::execute()}
Initially: TR ≠ ∅ {List of transition objects}
Initially: currST ≠ ∅ {Current state}
  tr = get_enabled_tr(TR)
  if type(tr) = preemptive then
    iterate_ref( tr )
    return true
  end if
  if fsm_ref( currST ) then
    iterate_fsm_ref( currST )
  end if
  if sdf_ref( currST ) then
    iterate_sdf_ref( currST )
  end if
  if type(tr) = nonpreemptive then
    iterate_ref( tr )
  end if
  return true
```

```
{fsm_model::postcondition()}
Initially: tr {Enabled transition}
Initially: currST {Current state}
  execute_commit_actions( tr )
  currST = next_state( tr )
  if reset(tr) then
    st_reset( curST )
  end if
  return true

{fsm_model::cleanup()}
clean_up()
return true

{fsm_model::iterate_ref( R )}
if fsm_ref( R ) then
  iterate_fsm_ref( R )
end if
if sdf_ref( R ) then
  iterate_sdf_ref( R )
end if
execute_choice_actions( R )
```

Figure 4: Simulation algorithm: Starting from a Heterogeneous Hierarchical FSM model

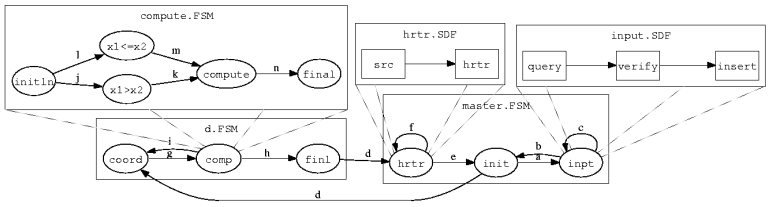


Figure 5: Behavioral decomposition of in-fill processor

the FSM and SDF MoCs. For enabling hierarchy, it is crucial that the graph infrastructure supports hierarchical graphs. Figure 2(a) shows the class diagram for the generic graph library.

Enabling SystemC with Hierarchical FSMs. Figure 2(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, respectively. 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 responsible for simulating that particular FSM.

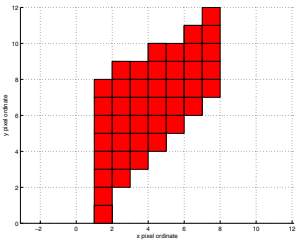
Enabling SystemC with Hierarchical SDFs & FSMs. The implementation of the SDF MoC once again takes advantage of the graph library and follows a similar approach as shown for hierarchical FSMs. Figure 2(c) shows the class diagram for the SDF library. The `sdf_block` class represents the computation block, which is connected by `sdf_edges`. The `sdf_graph` class 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_edges` because in SDF they represent FIFOs and all computation is modeled in the SDF blocks.

Enabling heterogeneous hierarchy between SDF and FSM models is shown using snippets of the class diagram in Figure 2(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 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.

5. EXAMPLE OF POLYGON IN-FILL PROCESSOR

This example shown in Figure 5 implements a variation of the polygon in-fill processor (PIP) from [2], 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 Figure 6(a).

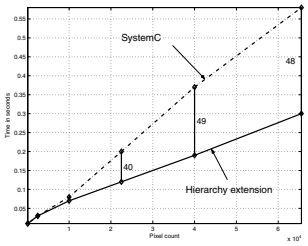
The PIP is a heterogeneous system composed of five behavioral components. Three of the components follow the FSM MoC and the other two adhere to the SDF MoC. The `master.FSM` is the master controller which constitutes of the `init`, `inpt` and `hrtr` states. The transitions between these states are guard/action pairs, which we annotate by a letter from the alphabet such as `a, b, c, d, ....` States `inpt` and `hrtr` are both refined with SDF sub-systems.. Transition `d` has an FSM refinement embedded as a `run-to-complete`. Similarly, `d.FSM` has three states includ-



(a) In-fill processor output

Pixels	SC(s)	SCH(s)	%Imp.
10^2	0.01	0.01	0
50^2	0.03	0.03	11
100^2	0.08	0.07	16
150^2	0.20	0.12	40
200^2	0.37	0.19	49
256^2	0.58	0.3	48

(b) Table showing simulation times



(c) Simulation results for in-fill processor

Figure 6: Polygon In-fill processor results

ing 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 Figure 5 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`.

6. 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. Figure 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). Figure 4 shows the tabulated values shown on the graph. We see that our version outperforms the SystemC model by approximately 50% (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.

7. CONCLUSION

In this paper we demonstrate how to design and implement extensions to SystemC in order to endow it with heterogeneous behavioral hierarchy. We show this with the addition of the hierarchical SDF MoC alongside the hierarchical FSM MoC. Addition of multi-MoC support for hierarchy is non-trivial and requires defining and understanding hierarchically compositional semantics across MoCs. We present our design and algorithms implementing the compositional simulation semantics which would enable SystemC CAD developers to implement these extensions. However, the formalization of the semantics is a subject of another paper. We use the PIP example to show a significant increase in simulation performance due to the preservation of behavioral hierarchy during simulation. Our future work entails addition of other MoC encapsulations in the hierarchy, as well as verification techniques that exploit the hierarchy. We also plan to release our implementation, as well as users guide on how to model heterogeneous systems with behavioral hierarchy to simplify designing complex embedded systems while gaining simulation speed as well.

8. REFERENCES

[1] A. Girault and B. Lee and and E. A. Lee. Hierarchical Finite State Machines with Multiple Concurrency Models. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 18, pages 742–760, 1999.

[2] Reinaldo A. Bergamaschi. *The Development of a High-Level Synthesis System for Concurrent VLSI Systems*. PhD thesis, University of Southampton, 1989.

[3] Boost. Boost graph library. Website: http://www.boost.org.

[4] E. A. de Kock, G. Essink, W. J. M. Smits, P. van der Wolf, J. -Y. Brunel, W. M. Kruijtzer, P. Lieverse and K. A. Vissers. YAPI: Application Modeling for Signal Processing Systems. In *In the proceedings of Design Automation Conference 2000*, 2000.

[5] Metropolis Group. Metropolis: Design environment for heterogeneous systems. Website: http://embedded.eecs.berkeley.edu/metropolis/index.html.

[6] D. Harel. Statecharts: A visual formalism for complex systems. In *Scientific Computing Program*, volume 8, pages 231–274, 1987.

[7] E. A. Lee and D. G. Messerschmitt. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. In *In Proceedings of IEEE Transactions on Computers*, volume Vol. C-36 of *NO. 1*, 1987.

[8] Edward A. Lee and Alberto L. Sangiovanni-Vincentelli. Comparing Models of Computation. In *In Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, pages 234–241. IEEE Computer Society, 1996.

[9] OCI. SystemC. Website: http://www.systemc.org.

[10] H. D. Patel and S. K. Shukla. *SystemC Kernel Extensions for Heterogeneous System Modeling*. Kluwer Academic Publishers, 2004.

[11] H. D. Patel and S. K. Shukla. Towards a heterogeneous simulation kernel for system level models: A systemc kernel for synchronous data flow models. In *IEEE Transactions in Computer-Aided Design*, volume 24, August 2005.

[12] H. D. Patel and S. K. Shukla. Towards behavioral hierarchy extensions for systemc. In *Forum on Design and Specification Languages (FDL '05)*, 2005.

[13] Ptolemy Group. Ptolemy II Website. http://ptolemy.eecs.berkeley.edu/ptolemyII/.

[14] SPECC. SpecC. Website: http://www.ics.uci.edu/specc/.

[15] SystemVerilog. System Verilog. Website: http://www.systemverilog.org/.