Ingredients for Successful System Level Automation & Design Methodology

Support for Multiple Models of Computation, Heterogeneous Behavioral Hierarchy, Model-driven Validation, and Service-oriented Tool Integration Environment

Hiren D. Patel

Dissertation submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Engineering

Dr. Sandeep K. Shukla, Chair
Dr. William T. Baumann, Committee Member
Dr. Stephen H. Edwards, Committee Member
Dr. Dong S. Ha, Committee Member
Dr. Michael S. Hsiao, Committee Member

April 6, 2007
Blacksburg, Virginia

Keywords: Models of Computation, Heterogeneity, Behavioral Hierarchy, Validation, Abstract State Machines, SystemC

Ingredients for Successful System Level Automation & Design Methodology

Support for Multiple Models of Computation, Heterogeneous Behavioral Hierarchy, Model-driven Validation, and Service-oriented Tool Integration Environment

Hiren D. Patel

(ABSTRACT)

This dissertation addresses the problem of making system level design (SLD) methodology based on SystemC more useful to the complex embedded system design community by presenting a number of ingredients currently absent in the existing SLD methodologies and frameworks. The complexity of embedded systems have been increasing at a rapid rate due to proliferation of desired functionality of such systems (e.g., cell phones, game consoles, hand-held devices, etc., are providing more features every few months), and the device technology still riding the curve predicted by Moore's law. Design methodology is shifting slowly towards system level design (also called electronic system level (ESL)). A number of SLD languages and supporting frameworks are being proposed. SystemC is positioned as being one of the dominant SLD languages. The various design automation tool vendors are proposing frameworks for supporting SystemC-based design methodologies. We believe that compared to the necessity and potential of ESL, the success of the frameworks have been limited due to lack of support for a number of facilities and features in the languages and tool environments. This dissertation proposes, formulates, and provides proof of concept demonstrations of a number of ingredients that we have identified as essential for efficient and productive use of SystemC-based tools and techniques. These are heterogeneity in the form of multiple models of computation, behavioral hierarchy in addition to structural hierarchy, model-driven validation for SystemC designs and a service-oriented tool integration environment. In particular, we define syntactic extensions to the SystemC language, semantic modifications, and simulation algorithms, precise semantics for model based validation etc. For each of these we provide reference implementation for further experimentation on the utility of these extensions.

This project received support from NSF, SRC and Bluespec.

Dedication

To those affected by 4/16/2007,

the Blacksburg and Virginia Tech. community,
the friends and families,
and
all my fellow Hokies.

– ***Hiren D. Patel***
4/19/2007, Nice, France.

Acknowledgments

I am extremely grateful to my advisor, Dr. Sandeep K. Shukla for his guidance, motivation, friendship, encouragement and support throughout my time at FERMAT.

I also thank Dr. Michael S. Hsiao, Dr. William T. Baumann, Dr. Stephen H. Edwards, and Dr. Dong S. Ha for serving as committee members for this dissertation. I would like to acknowledge the support received from the NSF CAREER grant CCR-0237947, the NSF NGS program grant ACI-0204028, an SRC Integrated Systems Grant and industrial grant from Bluespec Inc., which provided the funding for the work reported in this dissertation.

Special thanks are due to my roommates and friends from and around FERMAT: Nimal Lobo, Shatab Wahid, Syed Suhaib, Gaurav Singh, Deepak Mathaikutty, Sumit Ahuja, David Berner, and Animesh Patcha. Without forgetting the friends that have made Virginia Tech. and Blacksburg my home for many years, I'd like to extend my thanks to Ean Schiller, Mary Riddick, Chris Campbell, Sarah McKinney and Jenn DiCristina.

To my brother Anwar Jessa, and his beautiful wife Reem Jessa, never feel that I am far from you, wherever I am. To Caitlin Dougherty, thanks for always being there and inspiring me. To old friends, Mehjabeen Alarahkia, Masuma Rawji, Catherine-Rose Barretto and Nimit Khimji - miss you all. To all the others friends that have come and gone, thank you all.

To my surrogate family, Smita Manek, Anup Manek, Shruti Manek and Mitul Manek - thank you for being my family away from home. Finally, a heartfelt thanks goes to my parents Dhanji K. Patel and Maya D. Patel for their continued love, support, and encouragement in my endeavors.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The growing consumer demands for more functionality has lead to an increase in size and complexity of the final implementation of such designs. The semiconductor industry's ability to reduce the minimum feature sizes of integrated circuits has so far supported these growing consumer demands. This fast pace of increase in integration of devices per unit area on silicon, commonly known as the Moore's Law [1], is expected to continue up to the next ten years, roughly doubling the devices per chip every eighteen to twenty-four months [2]. However, even though current silicon technology is closely following the growing demands; the effort needed in modeling, simulating, and validating such designs is adversely affected. This is because current modeling tools and frameworks, hardware and software co-design environments, and validation and verification frameworks do not scale with the rising demands.

Fortunately, the electronic design automation (EDA) industry has historically played an integral role in providing engineers with the support for these challenging demands. For example, the introduction of register transfer level (RTL) as a higher abstraction layer [3] over schematics in traditional hardware description languages (HDL)s such as Verilog and VHDL. The RTL abstraction layer is nowadays accepted as the de-facto abstraction layer for describing hardware designs. However, in recent years, the EDA community is again pushing the abstraction layer a notch higher by promoting the electronic system level (ESL) layer for addressing the lack of scalable tools, frameworks and design methods. So far, the accepted definition of ESL is "a level above RTL including both hardware and software design" as suggested by The International Technology Roadmap for Semiconductors in 2004 [2]. However, there is a lack of consensus in the EDA community as to what this next level is to be.

A common prescription for an ESL design and verification methodology consists of a broad

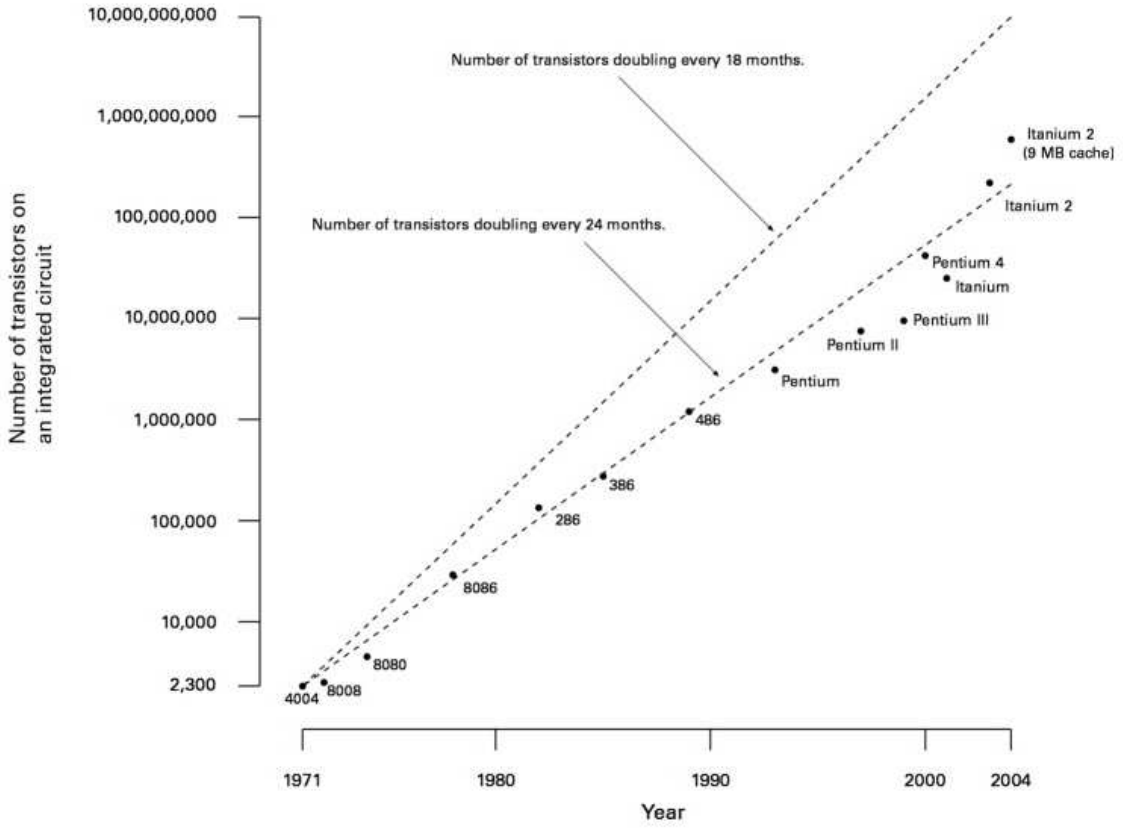


Figure 1.1: Moore's Law [1]

spectrum of environments for describing formal and functional specifications (we use modeling to denote this particular type of specification), hardware and software co-design models, architectural models, RTL and software models, and cell-level models. This prescription encompasses the modeling, simulation, validation and verification of system level designs. Models at the system level are descriptions above the RTL abstraction layer depicting the behavior of a system. Some members of the community suggest raising the abstraction layer [3, 4] for the purpose of modeling and simulation. But, there are a number of ways in which the abstraction layer may be raised above RTL. For example, Ptolemy II [5] presents heterogeneous models of computation (MoCs) for modeling and simulation of embedded software systems and Bluespec [6] uses a concurrent rule-based MoC for modeling, simulating and synthesizing hardware systems. SystemC [7, 8] introduces transaction level modeling and simulation, and Metropolis [9] uses a platform based and refinement-based approaches. ForSyDE [10] and SMLSys [11] use function-based semantics for the specification and simulation of heterogeneous systems.

These tools and frameworks provide alternative methods for specifying, simulating, validating and synthesizing today's complex designs, opposed to the use of traditional design flows

using HDLs. It is important for any ESL methodology to provide these methods [4]. This is to allow the ESL methodology to support engineers during design and validation. In this dissertation, we also attempt to address a few of these aspects. We concentrate on the issues of (1) functional specification (modeling), (2) simulation, (3) validation and (4) the integration of multiple tools. For (1) and (2), we focus on system level designs that exhibit heterogeneity and for (3), we center on a model-driven approach for validating system level designs. Point (4) concentrates on a method using dynamic services for integrating multiple tools. Note that (4) is not in the common prescription of an ESL methodology, but we deem this an important aspect because of the multitude of methods for modeling, simulation and validation that engineers will want to reuse. The system level design language (SLDL) with which we explore the four ESL methods is SystemC [7]. The motivation and importance of these four ESL methods and SystemC as our SLDL is succinctly described in the following subsections.

1.1.1 Why SystemC?

SystemC [8] is currently poised as a strong contender for a standard SLDL and with strong industrial backing; SystemC has become a part of many industrial design flows. It has been recently accepted as an IEEE standard [7]. The advantage of the underlying C++ language infrastructure and object-oriented nature extends usability further than any existing hardware description language at present. SystemVerilog [12] attempts at introducing some object-oriented concepts into their hardware description language, but generation of the object code require purchasing separately licensed compilers. On the other hand, SystemC provides embedded system designers with a freely available modeling and simulation environment requiring only a C++ compiler. SystemC is also a great medium for hardware and software co-simulation. Furthermore, C/C++ is a well accepted and a commonly used language by industry and academia alike, thus making its use even more compelling.

1.1.2 Modeling

Today's designs exhibit large scale heterogeneity in them. This means that different functionalities are contained within a single design and each of these functionalities is best described by distinct models of computation (MoC)s. A good example is an embedded system that usually integrates some system-on-chip (SoC) solution in the final implementation. The SoC may be required to provide functionalities for voice and data communication, music players, cameras and picture editing, all in one product. This is an example of heterogeneous design requirements that have a direct impact on the different components needed in the realization of the actual SoC design. Such an SoC may contain micro-controllers, memory blocks, on-chip busses for communication, digital signal processing elements for either video or music encoding and decoding, controllers to dictate the system interrupt priorities and

some event-based components to interact with the environment. These components may be best described using distinct MoCs such as finite state machines as good representations for controllers, data flow models for signal processing, communicating sequential processes for resource sharing and discrete-event for event-based computation.

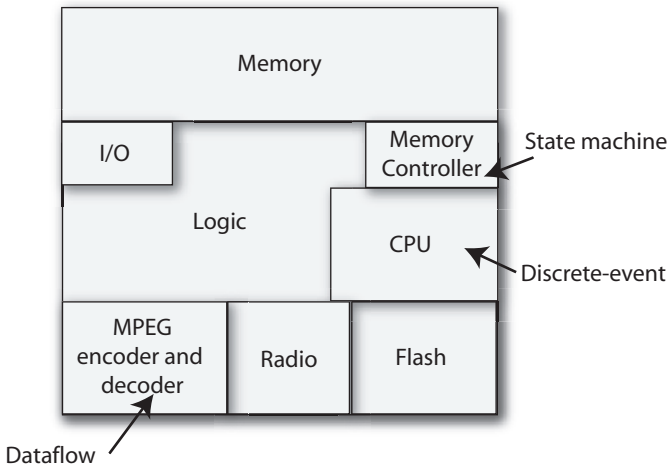


Figure 1.2: Example System-on-Chip Block Layout

Figure 1.2 shows a schematic example of an SoC that has components that follow a state machine behavior, a discrete-event behavior and a data flow behavior. However, the central problem with implementing such components is that traditional design methods do not have native support for this sort of heterogeneity. For example, hardware description languages (HDLs) such as VHDL [13] and Verilog [14] and some SLDLs such as SystemC and SystemVerilog [12] are used to describe hardware designs. However, during the modeling and simulation phases, VHDL and Verilog both follow discrete-event simulation semantics. So, any component that does not naturally follow the discrete-event MoC has to be mapped to its respective MoC. The workarounds required for correct functional specification of these heterogeneous components can be difficult and unintuitive [15]. Therefore, we see it important for SLDLs to support heterogeneous modeling via MoCs. This is the first ingredient we introduce for the modeling aspect of an ESL methodology.

1.1.3 Simulation

The simulation of heterogeneous designs is not trivial because each MoC has its own simulation semantics that describe the manner in which the executable entities of the design are simulated. There may be components embedded within each other that follow the same MoC, which we term *behavioral hierarchy*. Furthermore, there may be embeddings of components that follow different MoCs within another design component described using another MoC, which we call *heterogeneous behavioral hierarchy*. It is important that there is simulation

support for behavioral and heterogeneous behavioral hierarchy. Furthermore, an important aspect of behavioral hierarchy is preserving this hierarchy composition information for simulation. This can then be used to simulate different levels of hierarchy, whereby first the deepest levels are simulated followed by a level higher and so on. This is a way in which we can use the behavioral hierarchy information during simulation. This is unlike current HDLs and SLDLs that flatten all the behaviors in the system to the same level due to the inherent discrete-event MoC simulation semantics.

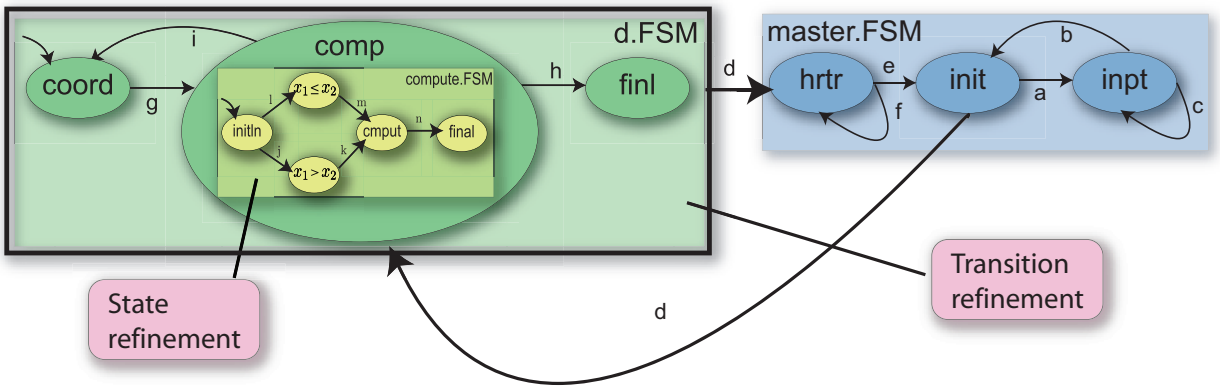


Figure 1.3: Example of Behavioral Hierarchy

Figure 1.3 shows an example of a heterogeneous behavioral hierarchy representation of a polygon infill processor [16]. The toplevel finite state machine is denoted by master.FSM. A transition refinement d.FSM exists for transition d. This refinement is another finite state machine that performs the infill processor computation. Within the d.FSM finite state machine, state comp has another finite state machine embedding to perform the computation of a line. Notice that this figure shows examples of state and transition refinements. This example is decomposed based on its behaviors and the appropriate embeddings are done in order to produce the polygon infill processor. So, for the simulation aspect of the ESL methodology, we provide kernel-level extensions for SystemC that support heterogeneous behavioral hierarchy simulation semantics. This is the second ingredient we provide.

1.1.4 Validation

The need for validation is straightforward. Models at any abstraction level need to be tested to ensure their correctness. The tests validate properties of the model. So, the objective of validation is to uncover inconsistences between the implementation behavior and the intended behavior. There are different test generation techniques, such as coverage-driven test methods, random test methods and manual test methods. Even though there are several tools that provide mechanisms for random test generation and coverage-driven test generation, manual test generation methods are also commonly used to drive the implementation

into different possible corner cases. In fact, manual test generation methods are used to supplement any of the other test generation methods. This is because designs may have certain safety-critical properties that need to be inidividually validated. Moreover, coverage-driven and random generation are used for a large coverage of the implementation and in general, a larger volume of tests. Directed tests are generated manually for only particular specific scenarios of the implementation.

The traditional approach for directing tests to reach a set of desired states in the implementation involves extracting the essential properties from the specification and then using these to manually create tests for the implementation. This is difficult, because validation engineers must have sufficient knowledge of the implementation to write tests that drive the inputs of the implementation to the desired states that validate or invalidate the property. Moreover, there may be multiple paths to the desired states, which makes the task of coming up with possible input sequences even harder. So, there seems to be few assisting methods that enable validation engineers to discover the sequence of inputs for directed tests to particular desired states. So, our work on the model-driven validation ingredient centers on enabling engineers with a method for generating input sequences that test specific corner case scenarios.

Related work using model-checking methods have been proposed to take the implementation as an input to create an abstract representation on which the corner case properties are verified using a model-checker [17, 18]. When a property holds in the abstract representation then it is said to hold in the implementation. However, other works [19, 20] have gone further by generating tests from counter examples. This is possible because whenever a property fails to verify, the diagnosis gives a counter example that can be used to steer the inputs of the implementation model to the failing property. However, the abstract representation loses information possibly causing particular tests to pass in the abstract representation and fail in the implementation. This can only happen if the abstraction is incorrect. This makes it important for the abstract representation to be executable.

Contrary to model-checking methods, a model-driven validation method requires validation engineers to first capture the essence of the design at a higher abstraction representation from the specification, after which an implementation can either be automatically generated or developed. Ideally, this abstract representation is formal and executable, providing possibilities for generating implementation directly from the abstract representation. Now, the properties can be validated in the abstract representation. However, we want to additionally use this abstract representation for generating tests. This can be done by exploring the specification such that an automaton representing the exploration is the result. Path traversals on the automaton can be used to generate tests for reachability of the desired states of the abstract representation. After generating the tests, they can be simultaneously executed in the abstract representation as well as the implementation. This facilitates the validation engineers with an approach for raising the abstraction layer for the initial modeling and simulation to the abstract representation, followed by a method for deriving inputs for corner case test scenarios. This model-driven validation method is our third ingredient.

1.1.5 Dynamic Integration of Multiple Tools

Most of the present SLDLs and SLDFs are primarily equipped with a modeling and simulation environment and lack facilities to ease model building, visualization, execution analysis, automated or directed test generation, and improved debugging, etc., which are becoming increasingly important to enhance the overall design experience. As evidenced by the release of commercial tools such as Debussy [21], ConvergenSC [22], VCS [23], Incisive [24] to mention a few, SLDLs themselves require additional supporting tools. However, with the multitude of commercial tools, an engineer must select the set of tools that best fits his/her needs and many times the existing tools do not suffice. This raises an issue of extendibility, where engineers have no way to extend the commercial tools or solutions to suit their own specific needs. Even if the SLDL can be subject to alterations, the abundance of open-source tools is subdued by the issue of efficient deployment mechanisms and re-usability. In addition to the ability of designers to extend a framework, they should also be able to cleanly interface and distribute their solution for integration with other SLDLs.

Some industrial outcomes to addressing these inadequacies are simulation-based dynamic validation frameworks such as ConvergenSC, VCS and Cadence Incisive. However each of these tackles various aspects of the design process. There are some overlapping features and some distinct desirable features, and at this moment it is not possible to unify the capabilities into one framework. Furthermore, the inherent lack of extendibility, disallows users to consider altering or integrating these different frameworks. This brings about the fourth ingredient of dynamic integration of multiple tools and design methods. We enable this by introducing the method of service-orientation for the dynamic integration of tools, which is our fourth ingredient. This is not commonly thought of as an enabling method for an ESL methodology, but we feel that it should be. This is because of the proliferation of ESL methods for specification, simulation and validation, but no method for reusing and integrating them in a clean manner.

The above mentioned sections individually motivate the need for the heterogeneous modeling, behavioral hierarchy and heterogeneous behavioral hierarchy simulation, model-driven validation for directed test case generation and service-orientation for dynamic integration of multiple tools ingredients. Each of these ingredients addresses important aspects of an ESL methodology. We address these four ingredients in this dissertation to augment the current methods for modeling, simulation, validation and dynamic integration of tools for a successful ESL methodology.

1.2 Problem Statement

This dissertation focuses on the four aspects of modeling, simulation, validation and dynamic integration of tools of an ESL methodology. We propose four ingredients that contribute

to each of these aspects and they are heterogeneous modeling, behavioral hierarchy and heterogeneous behavioral hierarchy simulation, model-driven validation for directed test case generation and service-orientation for dynamic integration of multiple tools for designs at the system level. The experimental SLDL that these four ingredients are geared towards is SystemC. Our proposal makes us ask the following questions:

- How can we both model and accurately simulate heterogeneous behaviors in SystemC so that we can represent the inherent heterogeneity in designs, and the hierarchical embeddings of these heterogeneous behaviors?
- How can we generate input sequences for validating corner case scenarios of a SystemC implementation by using an abstract representation to explore the possible input sequences and then using these input sequences to drive the implementation model.
- How can we enable dynamic integration of multiple third-party tools so that each tool acts as a service interacting with other services or clients through a common communicating meduim, and that these services can be dynamically turned on or off?

1.3 Organization

Chapter 2: Related work- We discuss some of the main research and industrial tools that provide interesting methodologies at ESL.

Chapter 3: Background- This chapter gives an overview of the necessary models of computation and information in better grasping the remainder of the chapters.

Chapter 4: Behavioral Hierarchy with Hierarchical FSMs- We introduce the notion of behavioral hierarchy with the FSM MoC. This chapter describes the simulation semantics and details on some of the implementation.

Chapter 5: Simulation Semantics for Heterogeneous Behavioral Hierarchy- This chapter extends on Chapter 4 by incorporating the SDF MoC such that hierarchical embedding of either FSM or SDF could be made to any level of depth. The simulation semantics have to be extended, which are also described in this chapter.

Chapter 6: Bluespec ESL and its Cosimulation with SystemC DE- The novel concurrent rule-based MoC extension for SystemC is described in this chapter. We also present the macro-based modeling language innovated. In addition, we present our investigation of the interoperability between Bluespec ESL MoC with SystemC discrete-event at the RTL abstraction.

Chapter 7: Model-driven Validation of SystemC Designs- We demonstrate our validation methodology by using Microsoft SpecExplorer for semantic modeling and validation using Abstract State Machines. This semantic model is then used for generating directed tests that can be simulated in the implementation model.

Chapter 8: Service-orientation for the Dynamic Integration of Multiple Tools- This chapter describes how we architect an environment for integrating multiple tools using middleware technology.

Chapter 9: Summary Evaluations- We describe our experience in employing the presented ingredients for ESL design and validation. This section also presents the limitations and advantages of our ingredients.

Chapter 10: Conclusion and Future Work- We sum up our experience with using the methodologies presented in this dissertation and provide insight into some of the possible future works from here.

Chapter 2

Related Work

2.1 System Level Design Languages and Frameworks

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 [5], Metropolis [9], YAPI [25] and SystemC-H [26]. We briefly describe some of these design frameworks employing MoCs and discuss how our attempt at integrating MoCs with SystemC differs from these design frameworks.

2.1.1 Ptolemy II

A promoter of heterogeneous and hierarchical designs for embedded systems is the Ptolemy II group at U. C. Berkeley [5]. The Ptolemy II project is a Java [27] implementation of a framework that focuses on embedded systems and in particular on systems that employ mixed technologies such as hardware and software, analog and digital, etc. Ptolemy II project is one of the renowned promoters of heterogeneous and hierarchical system design. Their experimentation with heterogeneity and hierarchy began with the Ptolemy Classic project where they introduce the FSM and SDF MoCs. However, this project was later abandoned and rejuvenated as the Ptolemy II project in Java. Ptolemy II follows an *actor-oriented* approach for modeling designs with a Java-based graphical user interface through which designers can "drag-and-drop" actors to construct their models. The two main types of actors are atomic actors and composite actors. The former describes an atomic unit of computation and the latter describes a medium through which behavioral hierarchy is possible. Ptolemy II's *directors* encapsulate the MoC behavior and simulate the model. A model or component that follows a particular director is said to be a part of that MoC's *domain*. Few of the domains implemented in Ptolemy II are component interaction, communicating sequential processes, continuous time, discrete-event, finite state machine, process networks, data flow,

and synchronous data flow. We give a brief description of some of the MoCs available when modeling using Ptolemy II.

Component Interaction (CI) The basis of the component interaction domain follows the construct for a data-driven or demand-driven application. An example of data-driven application is an airport departure and arrival display where information about flight departures, arrivals, delays, times, number, etc. are displayed for the public. Whenever a status changes of a particular flight, changes are made on the main server raising an event signaling the displays to perform a refresh or update.

Demand-driven computation on the other hand refers to the more common web-browsing environment where an HTTP request to a server acts as the demand resulting in an event on the server for a webpage. This event causes computation to return the requested webpage (given sufficient privilege for access) to the client requesting the webpage through a network protocol such as TCP.

Discrete-Event (DE) In the discrete-event domain, actors communicate through events on a real time line. An event occurs on a time stamp and contains a value. The actors can either fire functions in response to an event or react to these events. This is a common and popular domain used in specifying digital hardware. Ptolemy provides deterministic semantics to simultaneous events by analyzing the graph for data dependence to discover the order of execution for simultaneous events.

Finite State Machine (FSM) The FSM domain differs from the actor-oriented concept in Ptolemy II in that the entities in the domain are represented by states and not actors. The communication occurs through transitions between the states. Furthermore, the transitions can have guard conditions that dictate when the transition from one state to the other can occur. This domain is usually useful for specifying control state machines.

Process Network (PN) In the process network domain, processes communicate via message passing through channels that buffer the messages. This implies that the receiver does not have to be ready to receive for the sender to send, unlike the CSP domain. The buffers are assumed to be of infinite size. The PN MoC defines the arcs as a sequence of tokens and the entities as function blocks that map the input to the output through a certain function. This domain encapsulates the semantics of Kahn Process Networks [28].

Synchronous Data Flow (SDF) The synchronous data flow domain, operates on streams of data very suitable for DSP applications. SDF is a sub-class of DF except that the actor's consumption and production rates dictate the execution of the model. The actors are only fired when they receive the number of tokens defined as their consumption rate and expunge similarly only the number of tokens specified by the production rate. Moreover, static scheduling of the actors is performed to determine the schedule in which to execute the actors.

Even though Ptolemy II is one of the popular 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 [29]. The *charts 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. 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 or components of the 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++ opposed to Ptolemy II.

2.1.2 SystemC

SystemC [8] is an open-source modeling and simulation language released by the Open SystemC Initiative (OSCI). SystemC is basically a library of C++ classes that allows modeling at the register transfer level (RTL) and abstractions above RTL. Example of modeling at abstraction levels above RTL is transaction-level modeling (TLM), which is commonly used for fast design exploration and simulation [30]. SystemC's modeling framework consists of SystemC-specific constructs that help in describing the model in a programmatic manner. This program representation of the design is then simulated using SystemC's simulation framework, which follows a DE simulation kernel. A module instantiated using the `SC_MODULE()` macro in SystemC can express a set of SystemC process behaviors. The three types of SystemC processes are `SC_METHOD()`, `SC_THREAD()` and `SC_CTHREAD()`. These processes are associated with a member function called the *entry* function. When the process triggers, this entry function describes the internals of the process. Each of these processes can be triggered based on their sensitivity list. The SystemC process types are categorized as follows:

`SC_METHOD()`: Executes from the beginning to the end of the entry function without interruptions. Therefore, the computation in the entry function is atomic.

`SC_THREAD()`: The entry function associated with these types of SystemC processes usually contains an infinite loop with suspension points within it. The suspension points using `wait()` switch the context of the current process and yield control to the SystemC kernel. This allows for expressing concurrency in SystemC.

`SC_CTHREAD():` This process type is a derived type of the `SC_THREAD()` process type with the additional restriction that the process is sensitive to the clock edges. The entry function for this process type can also use the `wait()` command to suspend its execution, but it can also use `wait_until()`. Dynamic sensitivity utilizes the `wait_until()` construct.

Listing 2.1: SystemC Module Declaration

```
1 SC_MODULE( hello )
2 {
3   sc_in_clk clk;
4   void say_hello();
5   SC_CTOR( hello )
6     {
7       SC_METHOD( say_hello )
8   sensitive << clk.pos();
9     };
10 };
```

In SystemC, the notion of a component is defined by a SystemC module that is instantiated by using the `SC_MODULE()` macro. [Listing 2.1, Line 1] shows the instantiation of the `hello` module using the macro. This is followed by a declaration of a clock port using the `sc_in_clk` data-type in [Listing 2.1, Line 3]. Port declarations such as `sc_signal<>` can also be done here. The constructor requires the use of the `SC_CTOR()` as in [Listing 2.1, Line 5]. This macro expands to a constructor for the class `hello`. A SystemC module may have multiple SystemC processes instantiated but in our example we show the instantiation of one `SC_METHOD()` process in [Listing 2.1, Line 7]. This SystemC process is sensitive to the positive edge of the clock [Listing 2.1, Line 8]. The functionality associated with this process is implemented in the `say_hello()` entry function. There is an alternate approach to declaring a SystemC module which we do not explore here, but instead cite to the SystemC manual which explains the pure C++ way of describing this [8].

The implementation of the entry function `say_hello()` in this example is very simple as in Listing 2.2. The entry function simply prints out the message.

Listing 2.2: SystemC Module Implementation

```
1 #include "module.h"
2 #include <iostream>
3
4 void hello::say_hello()
5 {
6   std::cout << "Hello World" << std::endl;
7 }
```

The toplevel or `main()` function is shown in Listing 2.3. This toplevel shows the instantiation of the `hello` component, the binding to the clock and the start to simulation in [Listing 2.3,

Listing 2.3: SystemC Hello World main()

```
1 int sc_main (int argc , char *argv[]) {
2   sc_clock          clock;
3   hello helloComponent("HelloComponent");
4   helloComponent.clk( clock );
5   sc_start(clock, 10);
6   return 0;
7 }
```

Line 3], [Listing 2.3, Line 4] and [Listing 2.3, Line 5] respectively. This toplevel simulates this hello world example for 10 clock cycles.

For further details in modeling at different levels of abstraction using SystemC, we recommend readers to read the manuals and documents made available at the SystemC website [8].

2.1.3 SystemC-H

An experimental prototype developed to study heterogeneity for SystemC labeled SystemC-H [31, 32, 26] introduces three MoCs interoperable with SystemC 2.0.1. These three MoCs are FSM, SDF and communicating sequential processes (CSP). SystemC-H imposes stylistic guidelines to increase the modeling fidelity [33] of SystemC. Details with code examples and descriptions are given in [32]. We present a simple SDF model of the finite impulse response (FIR) to provide a brief overview regarding their modeling framework.

Figure 2.1: FIR Block Diagram

The approach employed by the authors of SystemC-H is very similar to how modeling in SystemC is performed. There are additional macros that assist in defining modules and a way for connecting them. For the SDF MoC, SystemC-H introduces the notion of SDF blocks and edges. The SDF blocks represent the functional blocks and the SDF edges are responsible for describing the production and consumption rates between each of the SDF blocks. This information is used for static scheduling of the design by the SDF kernel. We provide code snippets to show how these models are constructed using the language-level additions to SystemC for the extensions. In addition, we compare it with the original SystemC distribution's version of the FIR implementation to give a better idea of the similarities and differences.

The FIR example is modeled at a higher level of abstraction than an RTL model. The FIR model is separated into three functional blocks as shown in Figure 7.12. The Stimulus

Listing 2.4: `SC_MODULE()` Declaration

```
1 SC_MODULE(stimulus) {
2   sc_out<bool> reset;
3   sc_out<bool> input_valid;
4   sc_out<int>  sample;
5   sc_in<bool>  CLK;
6   sc_int<8>      send_value1;
7   unsigned cycle;
8   SC_CTOR(stimulus)
9   {
10      SC_METHOD(entry);
11      dont_initialize();
12      sensitive_pos(CLK);
13      send_value1 = 0;
14      cycle       = 0;
15  }
16  void entry();
17 };
```

Listing 2.5: `SDF_MODULE()` Declaration

```
1 extern sdf_graph sdf1;
2 SC_SDF_MODULE( stimulus ) {
3   SDFport<int> stimulusQ;
4   sc_int<8>      send_value1;
5   unsigned cycle;
6   SC_SDF_CTOR(stimulus)
7     {
8       SC_SDF_METHOD( entry, sdf1 );
9       send_value1 = 0;
10      cycle       = 0;
11    }
12  void entry();
13 };
```

Figure 2.2: Code Snippets for Stimulus Block

block generates inputs for the FIR, the FIR block performs the finite impulse response and the output block simply sends it to the standard output. We begin by describing how an `SC_MODULE()` declaration in SystemC looks when compared to an `SC_SDF_MODULE()` declaration, which is a macro from the SDF kernel extension.

The DE declaration of the Stimulus module in Listing 2.4 is straightforward, which has output and input ports, a constructor and the behavior of the Stimulus's `entry` function follows the `SC_METHOD()` process declaration. Compared to the declaration of the SDF module in Listing 2.5, we notice a similar structure where [Listing 2.5, Line 2] shows the declaration of `SC_SDF_MODULE()` followed by SDF specific ports using the `SDFport` type. The main difference is in the invocation of the process declaration. We add a macro `SC_SDF_METHOD()` [Listing 2.5, Line 8] that takes two arguments, the first being the entry function pointer and the second argument informs the SDF kernel that this module belongs to the SDF graph `sdf1`.

The entry function implementations for the Stimulus blocks are different mainly because they are realized by different MoCs shown in Listing 2.6 and Listing 2.7. Since the SDF MoC is an untimed MoC, we cannot implement the four cycle setup time for reset as done in DE's implementation for the reset, nor is it required since signal propagation is not necessary in an SDF MoC. However, given that a designer needs a cycle-accurate model, then a DE module embedding this SDF model can suffice the need for having to progress the simulation for four cycles before initiating the operation of the FIR. The only implementation required in the Stimulus block is pushing the data onto the SDF channel [Listing 2.7, Line 3]. For DE's Stimulus block, there are additional statements such as `input_valid.write()` that in essence provide the handshaking mechanisms for forwarding the control to the next module. For instance, [Listing 2.6, Line 10] shows a Boolean value `true` passed through the

Listing 2.6: DE Implementation

```
1 void stimulus::entry() {
2   cycle++;
3   if (cycle<4) {
4     reset.write(true);
5     input_valid.write(false);
6   } else {
7     reset.write(false);
8     input_valid.write( false );
9     if (cycle%10==0) {
10      input_valid.write(true);
11      sample.write( (int)send_value1
            );
12      send_value1++;
13    };
14  }
15 }
```

Listing 2.7: SDF Implementation

```
1 void stimulus::entry()
2 {
3   stimulusQ.push(send_value1);
4   send_value1++;
5 }
```

Figure 2.3: Code Snippets for Stimulus Block's Implementation

`input_valid` port. This informs the FIR block that an input is valid for further processing. The same signaling is done from the FIR to the Display block when the FIR block completes its computation on a data sample. This is not required in the SDF model because the executable schedule [33] is computed during initialization. Hence, in a similar fashion to the SDF Stimulus block, we implement the FIR and Display blocks, but do not show the code fragments for the sake of space. Instead we present the toplevel modules for the DE and SDF models in Listing 2.9 and Listing 2.8.

The `toplevel` module in Listing 2.9 constructs the SDF graph. The choice of the toplevel process can be of any SystemC type, but for presentation we select the `SC_METHOD()` process type. Assuming the SDF model is a component of a DE design, the toplevel's entry function must be manipulated according to the process type since they continue to follow SystemC DE semantics. In addition the master kernel is always the DE kernel in this example. The construction of the `toplevel` module is straightforward whereby pointers to SDF modules are declared and later initialized to their corresponding objects in the constructor. The important step is in constructing the SDF graph within the constructor (`SC_CTOR()`) since the constructor is only invoked once per instantiation of the object. The functions `set_prod(...)` and `set_cons(...)` set the arcs on the SDF graph with the production and consumption rates respectively. The arguments of these functions are: the SDF module that it is pointed to or from, the production or consumption rate depending on which function is called and the `delay`. We also enforce a global instantiation of `sdf_graph` [Listing 2.9, Line 5] type object for every SDF graph that is present in the model. Using the `schedule` class and the `add_sdf_graph(...)`, the SDF graph is added into a list that is visible by the extended SDF kernel [Listing 2.9, Line 33]. In addition, this toplevel process must have an entry function that calls `sdf_trigger()` [Listing 2.9, Line 15] signaling the kernel to process

Listing 2.8: DE's Toplevel

```
1#include <systemc.h>
2#include "stimulus.h"
3#include "display.h"
4#include "fir.h"
5SC_MODULE( toplevel )
6{
7   sc_in_clk clock;
8   fir * fir1;
9   display*  display1 ;
10  stimulus* stimulus1;
11  sc_signal<bool> reset;
12  sc_signal<int>   result;
13  sc_signal<bool>  input_valid;
14  sc_signal<int>   sample;
15  sc_signal<bool>  output_data_ready;
16
17  SC_CTOR( toplevel )
18    {
19      SC_METHOD( entry )
20  sensitive << clock.pos();
21
22        stimulus1 = new stimulus("
              stimulus_block");
23        stimulus1->reset(reset);
24        stimulus1->input_valid(
              input_valid);
25        stimulus1->sample(sample);
26        stimulus1->CLK(clock);
27        fir1 = new fir("process_body");
28        fir1->reset(reset);
29        fir1->input_valid(input_valid);
30        fir1->sample(sample);
31        fir1->output_data_ready(
              output_data_ready);
32        fir1->result(result);
33        fir1->CLK(clock);
34        display1 = new display("display
              ");
35        display1->output_data_ready(
              output_data_ready);
36        display1->result(result);
37    };
38  void entry()
39    { };
40};
```

Listing 2.9: SDF's Toplevel

```
1#include <systemc.h>
2#include "stimulus.h"
3#include "display.h"
4#include "fir.h"
5sdf_graph sdf1;
6SC_MODULE(toplevel) {
7   sc_in_clk clock;
8   fir * fir1;
9   display*  display1 ;
10  stimulus* stimulus1;
11  SDFchannel<int> stimulusQ;
12  SDFchannel<int> firQ;
13
14  void entry_sdf()  {
15    sdf_trigger(name());
16  }
17  SC_CTOR(toplevel) {
18    SC_METHOD(entry_sdf);
19    sensitive << clock;
20    fir1 = new fir("process_body*");
21    display1 = new display("display")
          ;
22    stimulus1 = new stimulus("
          stimulus");
23    stimulus1->stimulusQ(stimulusQ);
24    fir1->stimulusQ(stimulusQ);
25    fir1->firQ(firQ);
26    display1->firQ(firQ);
27    stimulus1->set_prod(fir1, 1 , 0);
28    fir1->set_cons(stimulus1, 1, 0);
29    fir1->set_prod(display1, 1, 0);
30    display1->set_cons(fir1, 1, 0);
31    display1->set_prod(stimulus1, 1,
          1);
32    stimulus1->set_cons(display1, 1,
          1);
33    schedule::add_sdf_graph(name(), &
          sdf1);
34  }
35};
```

Figure 2.4: Code Snippets for Toplevel

all the SDF modules corresponding to this toplevel SDF module. These guidelines enable the modeler to allow for heterogeneity in their models since the toplevel process can be sensitive to any signal that fire the SDF model.

Focusing only on the SDF MoC so far, we realize that declaring the SDF modules for this example are similar to declaring modules using the DE MoC with a few differences in port types and macros used. The main distinction between these two MoCs is the implementation of the modules and the construction of the SDF graph. From Listing 2.6 and Listing 2.7 we see that the SDF module only implements the core functionality of the block, not requiring any synchronization such as signaling the next block that its computation is completed, nor a handshake to ensure that the data is available on the channel. Furthermore, the execution schedule of the SDF graph is computed during initialization notifying the user of a poorly constructed SDF model before starting the simulation. As for constructing the SDF graph, it is necessary to describe the structure of the model, which may also contain cyclic directed graphs [31, 33, 32]. Though SystemC-H supports heterogeneity, it lacks heterogeneous behavioral hierarchy, disallowing designers to hierarchically compose designs with varying MoCs. In addition, SystemC-H requires significant alteration to the original SystemC scheduler, which we attempt to improve on. We understand that we can represent MoCs and their interactions as simple libraries which can be linked with SystemC models without altering the standard reference implementation.

2.1.4 HetSC

Another approach to introducing heterogeneity in SystemC is via the HetSC [34, 35, 36] library, which is basically a library of communication channels for SystemC. The clear separation between the communication and the computation in SystemC enables designers to create specific channels which in the HetSC library's case, are MoC-specific channels. The authors of HetSC create a library of channels allowing the modeling of the PN, CSP, SDF and SR MoCs using their hierarchical and primitive channels. The advantage of this approach is that the SystemC DE semantics are used as an underlying mechanism to simulate these MoCs, thus there are no issues with interoperability and interfacing with SystemC. However, the resulting simulation efficiency with this approach is unknown as it is not investigated yet. We suspect that the MoCs using HetSC will perform slower than kernel-level extensions.

2.1.5 YAPI

YAPI is a C++-based run-time signal processing programming interface to construct models using the Kahn process networks (KPN) MoC [25]. The purpose of YAPI is to primarily allow re-usability of signal processing applications for hardware and software co-designs. This programming interface follows a homogeneous MoC, primarily the KPN MoC. The main disadvantage for system designers is that systems nowadays are heterogeneous with

more behavioral components than just signal processing cores. We understand that the purpose of YAPI is specific to signal processing applications, hence, on its own, it does not suffice the need for a multi-MoC framework for heterogeneous and hierarchical designs.

2.1.6 Metropolis

Another U. C. Berkeley group works on a project called Metropolis [9], whose purpose is again directed towards the design, verification and synthesis of embedded software. However, their approach is different than that of Ptolemy II's. Metropolis has a notion of a meta-model 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.

2.1.7 ForSyDE and SML-Sys

ForSyDe is a library-based implementation that provides a computational model for the synchronous domain with interfaces implemented in Haskell [37]. However, since ForSyDe has a single computational model based on the synchrony assumption, which is best suited for applications amenable to synchrony. SML-Sys when compared with ForSyDe has a higher modeling fidelity [32], since it is a multi-MoC modeling framework based on the generic definition in [38], which is an extension of the ForSyDe methodology. ForSyDe illustrates the formulation of the synchronous MoC and SML-Sys takes this a few steps further and formulates the untimed, clocked synchronous and timed MoCs. ForSyDe's computational model is based on the synchrony assumption and is limited to applications amenable to synchrony. The SML-Sys framework has an untimed MoC that expresses dataflow models and state machines with ease, a clocked synchronous MoC that models digital hardware and finally a timed MoC that models real-time requirements like timing analysis to compute clock cycle width.

2.1.8 Bluespec BSC

Bluespec [6] released the Bluespec compiler (BSC) that generates synthesizable Verilog. The input to BSC is a rule-based MoC based on term rewriting systems. The basic entities of Bluespec are modules, rules, methods, interfaces and Bluespec data-types. The strength of their approach involves dealing with concurrent and shared-state modeling [39, 40, 41]. Since, the modeling language is built on Verilog, the behaviors of every rule or method is essentially similar to Verilog. Another attractive characteristic of Bluespec is that there is a path from the model using the rule-based MoC to synthesizable Verilog.

2.1.9 API and PLI Approaches

Co-simulation of models in Verilog and C/C++/SystemC models often employ Verilog programming language interface (PLI). Verilog enables this by supporting invocations of C functions. Similarly, application programming interface (API) provides an interface for VHDL and Verilog to invoke C functions. These are ways in which HDLs can connect to C/C++ models. The API or PLI approach is commonly used when models of certain IPs already exist in traditional HDLs such as Verilog and VHDL that are to be integrated into a model at the system level. Instead of implementing the functionality of the IP or component in the SLDL, the API or PLI approach allows connection between the two modeling frameworks.

2.1.10 Term Rewriting Systems

A Term Rewriting System (TRS) is a reduction system [42] in which rewrite rules are applied to terms to create new terms. Terms are constructed from variables, constant symbols and function symbols applied to terms. The rewrite rules usually are of the form $x \rightarrow y$, where x and y are terms, x is not a variable, and every variable from y occurs in x as well. Given a term r, one has to first find a rule of the form $x \rightarrow y$ such that there is a unifier θ for r and x. (A unifier of two terms is an assignment of variables in both the terms such that after applying the assignments, the two terms become the same.) Given a unifier θ, applying this rule to term r reduces it to $y\theta$. $y\theta$ represents the term obtained by replacing the variables of the term y by the assignments in θ. If θ is a unifier of a subterm r' of r and x, then application of this rule would reduce r to a term where r' part of r is replaced by $y\theta$.

TRS are often used to give semantics to programming languages, and also to code axiomatic systems and proof rules to encode theorem provers. TRS can also be used to describe parallel and asynchronous systems because rule application can be thought of as computation step, and multiple rules can be applied in parallel to change terms to mimic parallel computation. Bluespec originated from this idea of TRS based "operation centric" hardware description, where the state of the system can be viewed as terms and computation steps can be viewed as application of rewrite rules. Therefore, a hardware system can be modeled using a set of

rewrite rules very similar to the operational semantics rules [43, 41].

2.1.11 TRS Modeling

In the approach based on hardware synthesis from "operation centric" descriptions [43], the behavior of a system is described as a collection of guarded atomic actions or rules, which operate on the state of the system. Each rule specifies the condition under which it is enabled, and a consequent allowable state transition. The atomicity requirement simplifies the task of hardware description by permitting the designer to formulate each rule as if the rest of the system is static. Using this methodology, any legitimate behavior of the system can be expressed as a series of atomic actions on the state.

As an example of "operation centric" descriptions, a two-stage pipelined processor, where a pipeline buffer is inserted between the fetch stage and the execute stage, is described here. A bounded FIFO of unspecified size is used as the pipeline buffer. The FIFO provides the isolation to allow the operations in the two stages to be described independently. Although the description reflects an asynchronous and elastic pipeline, the synthesis can infer a legal implementation that is fully-synchronous and has stages separated by simple registers.

"Operation centric" description framework uses the notation of Term Rewriting Systems (TRS) [43]. A two-stage pipelined processor can be specified as a TRS whose terms have the signature, *Proc (pc, rf, bf, imem, dmem)*, where, *pc* is the program counter, *rf* is the register file (an array of integer values), *bf* is the pipeline buffer (a FIFO of fetched instructions), *imem* is the instruction memory (an array of instructions), and *dmem* is the data memory (an array of integer values).

$rf[r]$ gives the value stored in location r of rf, and $rf[r := v]$ gives the new value of the array after location r has been updated by the value v. Now, the instruction fetching in the fetch stage can be described by the rule:

Fetch Rule:

Proc(pc, rf, bf, imem, dmem)

$\longrightarrow$ Proc(pc+1, rf, *enq*(bf, imem[pc]), imem, dmem)

The execution of the different instructions in the execute stage can be described by separate rules. The *Add* instruction can be described by the rule:

Add Exec Rule:

Proc(pc, rf, bf, imem, dmem) where Add(rd,r1,r2)=*first*(bf)

$\longrightarrow$ Proc(pc, rf[rd:=v], *deq*(bf), imem, dmem) where v=rf[r1]+rf[r2]

The *Fetch* rule above fetches instructions from consecutive instruction memory locations and enqueues (using *enq()*) them into *bf*. The Fetch rule is not concerned with what happens if the pipeline encounters an exception. On the other hand, the *Add Exec* rule processes the next pending instruction in *bf* as long as it is an *Add* instruction.

In this pipeline description, the Fetch rule and an execute rule can be ready to fire simultaneously. Although, conceptually, only one rule should be fired in each step, an implementation of this processor description must carry out the effect of both rules in the same clock cycle. The implementation does not behave like a pipeline without concurrent execution. However, the implementation must also ensure that a concurrent execution of multiple rules produces the same result as a sequential order of execution. Thus, in hardware synthesis from "operation centric" descriptions, detecting and scheduling valid concurrent execution of rules is the central issue [39, 43, 41].

2.2 Verification of SystemC Designs

Recent work in design and verification methodologies for SystemC using SpecExplorer are presented in [44, 45, 46]. In their approach, the design begins with UML specifications for the SystemC design and PSL properties. Both the SystemC and PSL UML specification are translated into ASM descriptions and simulated using SpecExplorer's model simulator. [46] suggests that they also implement an AsmL specification of SystemC's simulation semantics [44], however no executable is available to check their correctness. This simulation provides a first stage of verification that ensures that the PSL properties specified hold in the ASM specification of the SystemC design. This SystemC ASM specification is then translated into a SystemC design using C++ and the PSL properties in C#. These two are then compiled and later executed together to verify whether the properties are satisfied at the low-level implementation. Unfortunately, none of the implementations of the SystemC simulation semantics, the translation tools from UML to AsmL, or AsmL to SystemC/C# are accessible and available for us to reuse or check validity of their claims.

In summary, their work uses the modeling, simulation and automata generation capabilities of SpecExplorer but not the test case generation and execution tools. They only hint towards the possibility of using it for test generation in their overall design flow. In [46], the same authors present algorithms for generating FSMs from SystemC source code. They also claim to have translation tools to convert the extracted FSM into AsmL specification. These algorithms for FSM generation use similar concepts such as state grouping as the ones in SpecExplorer and once again the authors hint that this can be used for conformance testing and equivalence checking, but this is not presented. Their work attempts at providing a top-down and bottom-up flow for verification of SystemC designs mainly focusing

on PSL assertion verification, but not test case generation and execution for validation purposes. The main distinction of their work and our work is that we do not convert ASMs to SystemC or SystemC to ASMs. Instead, we promote a model-driven approach where the semantic model is done first for the correctness of the functionality and conformance to the natural specification. This is followed by an implementation model in SystemC developed independently. Then, the conformance between the semantic and the implementation model is validated by generating tests in SpecExplorer and executing them in both the semantic and the implementation model. This is how the work in this paper distinguishes itself from the works mentioned in [45, 46, 44].

Another work on functional verification of SystemC models is proposed in [17]. In general, an FSM is generated by performing static analysis on the source code very much like [46] and this FSM is used to generate test sequences for the system under investigation. Authors of [17] use an error simulation to inject errors into a model that is compared with an error-free model for detecting possible errors. The biggest difference in the approach described in [17] is the lack of control a designer has in directing the test case generation. For instance, the final states (accepting states for us) are not defined by the designer. In addition, these authors use static analysis to parse SystemC and generate extended FSMs, but they do not provide formal semantics for SystemC. This is important to check whether the abstract model in extended FSMs is a correct representation of the SystemC design. Our work on the other hand focuses on providing designers with the capability of defining the exact states of interest and only generating input sequences up to those states. This is done by creating two independent models, first the semantic then the implementation. Then the semantic model is used for directing tests in the implementation model.

Authors of [18] propose labeled Kripke structure based semantics for SystemC and predicate abstraction techniques from these structures for verification. They treat every thread as a labeled Kripke structure and then define a parallel composition operator to compose the threads. They also provide formal semantics to SystemC. Our work differs from this work in the same way that of [17] that we provide a model-driven approach. The authors of [18] create their abstract model from the implementation. Moreover, they do not present any algorithms for traversing the parallely composed Kripke structures for test generation.

The authors in [47] presented the ASM-based SystemC semantics that was later augmented for the newer versions of SystemC by [45]. However, the original ASM semantics in [47] did not present any support for test case generation and validation from ASMs. Also, no fully executable versions of these are available which is surprising given that the main idea of using ASMs for providing semantics is that executable versions could be made available for reproducibility elsewhere.

2.3 Reflection and Introspection

Introspection is the ability of an executable system to query internal descriptions of itself through some *reflective* mechanism. The reflection mechanism exposes the structural and runtime characteristics of the system and stores it in a data-structure. We call data stored in this data-structure *meta-data*. This data-structure is used to query the requested internal characteristics. The two sub-categories of the reflection meta-data are structural and runtime. Structural reflection refers to descriptions of the structure of a system. For SystemC, structural reflection implies module name, port types and names, signal types and names, bitwidths, netlist and hierarchy information. On the other hand runtime reflection exposes dynamic information such as the number of invocations of a particular process, the number of events generated for a particular module and so on. An infrastructure that provides for R-I (either structural or runtime reflection) is what we term an reflection service.

2.3.1 Existing Tools for Structural Reflection

Several tools may be used for implementing structural reflection in SystemC. Some of these are SystemPerl [48], EDG [49], or C++ as in the BALBOA framework [50] and Pinapa [51]. However, each of these approaches have their own drawbacks. For instance, SystemPerl requires the user to add certain hints into the source file and although it yields all SystemC structural information, it does not handle all C++ constructs. EDG is a commercial front-end parser that parses C/C++ into a data-structure, which can then be used to interpret SystemC constructs. However, interpretation of SystemC constructs is a complex and time consuming task, plus EDG is not available in the public domain. BALBOA implements its own reflection mechanism in C++ which again only handles a small subset of the SystemC language. Pinapa is a new front-end for SystemC that offers an intrusive solution for parsing SystemC by altering GCC and SystemC's source code. As for runtime reflection, to our knowledge, there is no framework that exposes runtime characteristics of SystemC models.

2.3.2 Edison Group's C/C++ Front-end (EDG)

One of the first successes in parsing SystemC to retrieve structural and behavioral information was possible by the C/C++ front-end developed by the Edison Group. This EDG front-end parses C/C++ and generates a complex intermediate representation with full expansion of the code. This intermediate representation is available as a pointer to the structure for further analysis, or can also be dumped to a file in some specific proprietary format. For successful extraction of SystemC constructs, multiple traversals through the intermediate data-structure are made. This is a common solution employed in many products that require extracting information from SystemC. The downside of this approach is that it is a commercial tool.

2.3.3 Pinapa

Pinapa [51] is an open-source SystemC front-end that uses GCC's front end to parse all C++ constructs and infer the structural information of the SystemC model by adding a hook to SystemC's elaboration phase. This idea of executing the elaboration phase in order to extract information about the specific design such as the number of instances of a component, the connectivity and so on, is a very attractive solution. SystemC's elaboration constructs all the necessary objects and performs the bindings after which a regular SystemC model begins simulation via the `sc_start()` function. Instead, Pinapa examines the data-structures of SystemC's scheduler and creates its own internal representation. Once the internal representation is generated, the SystemC model executes. This is a very good solution for tackling the SystemC parsing issue. However, Pinapa employs an intrusive approach that requires modifications of the GCC source code which make it (i) dependent on changes in the GCC code-base, and (ii) forbids the use of any other compiler with the benefit of parsing SystemC. In addition, a small change to the SystemC source is also required.

2.3.4 SystemCXML

SystemCXML is an attempt at parsing SystemC designs using a suite of open-source tools. The tools used in SystemCXML are Doxygen [52], Apache's Xerces-C++ library [53] and C++ programs. Doxygen is used as a front-end parser for C/C++ and using Doxygen's preprocessor, the SystemC-specific macros are identified by supplying an additional input file with all the types and macros defined. The SystemC source files are parsed using Doxygen and the output is spit out in XML files. No special tagging is done in this process other than identifying C/C++ constructs with the built-in C/C++ parser in Doxygen. Therefore, it is necessary to actually parse the relevant SystemC constructs from this representation. Using Xerces-C++, the authors of SystemCXML parsed the output from Doxygen and created an intermediate abstract system-level description (ASLD) that represented the SystemC design in an XML format. Additional modules were added to build netlist information and visualization from this representation.

2.3.5 SystemPerl

Veripool's SystemPerl implemented in the `SystemC::Parser` [48] module implements a SystemC parser and netlist generator using Perl scripts. Using the power of regular expressions, the task of recognizing SystemC constructs is made easy, and the open-source nature of SystemPerl makes its use much more attractive. However, one major distinction between EDG and System-Perl is that SystemPerl only extracts structural information and not behavioral. For most uses, structural information is sufficient, unless considering synthesis from SystemC which requires all the behavioral information as well. To our understanding, SystemPerl has

some limitations. One of them is that it requires source-level hints in the model for the extraction of necessary information and the internal representation of SystemPerl cannot be easily adapted to other environments and purposes.

2.3.6 Karlsruhe SystemC Parser Suite

Abbreviated to KaSCPar is the Karlsruhe SystemC Parser [54] suite designed to parse SystemC 2.1 keywords by tokenizing the structural and behavioral aspects of the language. The authors implement KaSCPar using Java [27] and JJTree [55] providing the SystemC community with a two component tool. The first component generates an abstract syntax tree (AST), which is an elaborated description of the SystemC model or design in XML format. This component is compiled into the binary `sc2ast` and upon its execution on a SystemC design, the C/C++ syntax along with SystemC constructs are both parsed into tokens. The second component uses the `sc2ast` to first generate the AST and then use the AST to generate XML description of the SystemC design.

2.3.7 ESys.NET Framework and Introspection in SystemC

ESys.NET [56] is a system level modeling and simulation environment using the .NET framework and C# language. This allows ESys.NET to leverage the threading model, unified type system and garbage collection along with interoperability with web services and XML or CIL representations. They propose the managed nature of C# as an easier memory management solution with a simpler programming paradigm than languages such as C or C++ and use the inherent introspective capabilities in the .NET framework for quicker debugging. They also employ the common intermediate language (CIL) as a possible standard intermediate format for model representation. One of the major disadvantages of using the .NET framework is that it is platform dependent. The .NET framework is primarily a Microsoft solution making it difficult for many industries to adopt technology built using the .NET architecture because of well-established Unix/Unix-variant industrial technologies.

There are obvious advantages in making ESys.NET a complete mixed-language modeling framework interoperable with SLDLs such as SystemC. However, we see no easy solution for interoperability between managed and unmanaged frameworks partly because integrating the unmanaged project in a managed project reduces the capabilities of the .NET architecture. For example, mixing managed and unmanaged projects does not allow the use of .NET's introspection capabilities for the unmanaged sections of the project. A natural way to interact between different language paradigms and development approaches (managed versus unmanaged) is to interface through a service oriented architecture. Microsoft has their own proprietary solution for this such as COM, DCOM and .NET's framework. Unfortunately, one of the major drawback as mentioned earlier is that C# and .NET framework is proprietary technology of Microsoft. Even though there are open-source attempts at imitating

C#, the .NET framework as a whole may be difficult to conceive in the near future [57].

The authors of [56], inspired by the .NET framework's reflection mechanism propose the idea of a composite design pattern for unification of data-types for SystemC. They enhance SystemC's data-type library by implementing the design pattern with additional C++ classes. This altered data-type library introduces member functions that provide introspection capabilities for the particular data-types. However, this requires altering the SystemC data-type library and altering the original source code to extract structural information. This raises issues with maintainability with version changes, updates and standard changes due to the highly coupled solution for introspection.

A different approach for exposing information about SystemC models to graphical user interfaces (GUI) is described in [58]. This work describes a methodology for interfacing SystemC with external third party tools where they focus on a GUI value-change dump viewer as the external tool. This requires allowing the VCD viewer to probe into the SystemC model and display the timestamp, the type of the signal and the current value of the signal. The authors document the required changes to the SystemC scheduler `sc_simcontext`, `sc_signal` and `sc_signal_base` classes along with their additional interface classes to expose the type of a SystemC signal and its corresponding value. They implement the observer pattern such that the VCD viewer accepts the messages from the altered SystemC source and correctly displays the output.

2.3.8 BALBOA Framework

The BALBOA [50] framework describes a framework for component composition, but in order to accomplish that, they require R-I capability of their components. They also discuss some introspection mechanisms and whether it is better to implement R-I at a meta-layer or within the language itself. We limit our discussion to only the approach used to provide R-I in BALBOA.

BALBOA uses their BIDL (BALBOA interface description language) to describe components, very similar to CORBA IDLs [59]. Originally IDLs provide the system with type information, but BALBOA extends this further by providing structural information about the component such as ports, port sizes, number of processes, etc. This information is stored at a meta-layer (a data-structure representing the reflected characteristics). BALBOA forces system designers to enter meta-data through BIDL, which is inconvenient. Another limitation of this framework is that the BIDL had to be implemented. Furthermore, the designer writes the BIDL for specifying the reflected structure information which can be retrieved automatically from SystemC source. BALBOA also does not perform runtime reflection.

2.3.9 Meta-model Based Visual Component Composition Framework (MCF)

The MCF framework takes ideas from the BALBOA framework regarding IP component composition and revamps the efforts by incorporating several new methodologies. MCF supports composition of RTL with RTL, transaction-level (TL) with TL and mixed abstractions of RTL with TL. The authors of MCF provide a visual modeling language for easing the task of component composition. They use a generic modeling environment (GME) [60] framework to describe their visual language. GME is a meta-modeling framework in which the authors implement a component composition meta-model. The advantage of this approach is that semantic and syntactic checks are performed on-the-fly as the designer instantiates a model and begins composing or creating the design. Upon completion of the design, a model targeted to SystemC can be generated. To be able to do this, they implement a reflection and introspection module using KaSCPar [54]. They use the reflected information about the IPs and designs to explore the composibility of the IPs. This includes type checking, composition based on behavioral types and untyped components.

2.3.10 Java, C# .NET Framework, C++ RTTI

Here, we discuss some existing languages and frameworks that use the R-I capabilities. They are Java, C# and the .NET framework and C++ RTTI. Java's reflection package `java.lang.reflect` and .NET's reflection library `System.Reflection` are excellent examples of existing R-I concept implementations. Both of these supply the programmer with similar features such as the type of an object, member functions and data members of the class. They also follow a similar technique in providing R-I, so we take the C# language with .NET framework as an example and discuss in brief their approach. C#'s compiler stores class characteristics such as attributes during compilation as meta-data. A data-structure reads the meta-data information and allows queries through the `System.Reflection` library. In this R-I infrastructure, the compiler performs the reflection and the data-structure provides mechanisms for introspection.

C++'s runtime type identification (RTTI) is a mechanism for retrieving object types during execution of the program. Some of the RTTI facilities could be used to implement R-I, but RTTI in general is limited in that it is difficult to extract all necessary structural SystemC information by simply using RTTI. Furthermore, RTTI requires adding RTTI-specific code within either the model, or the SystemC source and RTTI is known to significantly degrade performance.

2.4 Service-orientation

2.4.1 Adaptive Communication Environment (ACE) and the ACE Orb Way (TAO)

ACE [61] is a library of C++ classes geared towards making the task of implementing communication software easy via reusability and patterns for concurrent communication. This toolkit helps with the development of portable and high-performance applications. Especially for applications that are network or multi-threaded oriented. A clear advantage of the ACE toolkit is its implementation of various popular design patterns [62]. This makes implementation of many of the concurrency models very simple.

TAO [63] is a CORBA [64] implementation using ACE as its underlying toolkit. It provides an API to implement distributed applications that communicate over an ORB. The idea is to have various distributed services that interact with each other along with other clients through the ORB.

2.4.2 Service-oriented Software

Many distributed applications use middleware such as CORBA [59] to integrate a system with services and floating objects accessible via ORB. System level design languages can take advantage of middleware for co-simulation purposes as shown in [65]. [66] discusses a co-simulation environment for SystemC and NS-2 [67] which can also be integrated into CARH with relative ease. In addition effective testing and parallel simulation execution is viable as demonstrated by [68]. CARH utilizes the TAO & ACE environments to provide a service oriented architecture extendable for cosimulation, distributed testing and any user-desired services.

Chapter 3

Background

3.1 Fidelity, Expressiveness and Multiple Models of Computation

The term *Model of Computation* (MoC) is a frequently used term in the Computer Science literature, in enough diverse contexts so as to arise confusion. As a result, it is important to clarify the context in which we use the term, and also draw the reader's attention to the other alternative usages of this term. Of course, contextual variety does not mean the term is overloaded with distinct interpretations. The invariant in all of its usage archaic or modern, is that it refers to a formalized or idealized notion of computing rules. For example, going back centuries, Euclid's algorithm for finding the greatest common divisor of two numbers can be idealized to a pencil and paper model of computation. There, the only steps allowed were multiplication followed by subtraction, followed by comparison and repetitions of this sequence of steps until a termination condition is reached. Our definition of an MoC describes the manner in which the communication and the computation occur in that particular MoC. However, 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 [5] and SystemC-H [26]. Following that, a framework's *fidelity* is the capability of the framework to faithfully model an MoC. This introduces the idea of frameworks supporting multiple MoCs, thus heterogeneity.

The multi-MoC framework in Ptolemy II requires the user to build the components in Java, using some characteristics of the target MoC domain, and then placing the components in the appropriate domain under the control of a domain director, which schedules and guides the simulation of these components, and helps the components communicate according to domain specific communication rules. In ForSyDe [10], the components are built using functional programming language Haskell, suitable for the denotational mode of expressing the computational structure of the components. Recently, the same denotational MoC

framework has been implemented [69] in another functional programming language SML. The facilities of multi-MoC modeling provided by these frameworks are designed in a way, so that the multi-MoC designs can be faithfully modeled and simulated. However, as the frameworks vary in the granularity of the MoC domains supported, one can imagine that the fidelity of the framework varies. For example, in Ptolemy II, an SDF model is untimed, and if a DSP application needs to compute over multiple cycles, it is less direct to model such a behavior in the SDF domain of Ptolemy II. Similarly, in the ForSyDe or similar denotational framework, if one wants to model rendezvous style communication paradigm, it is usually not a part of the MoC classification in such frameworks. As a result, one has to approximate it by indirect means.

Another aspect of such frameworks is *expressiveness*. When a modeling framework does not impose particular structures or specific process constructors and combinators for modeling specific MoCs, but rather provides the user with the full expressiveness of a programming language to model systems according to the user's own modeling style and structure, the expressiveness is high. However, if the framework does not allow the user to make use of all possible ways that a standard programming language permits, but rather imposes structural and stylistic restrictions through process constructors and combinators, then such a framework has lower expressiveness than standard programming language. However, this lower expressiveness usually is matched with high fidelity, because according to this notion of expressiveness, a low expressiveness in a framework implies that it provides the user with strict structures and guidelines to faithfully reflect an intended MoC, hence having high fidelity.

The question then arises as to what the compromises between high fidelity/lower expressiveness and low fidelity/high expressiveness are. The lower expressiveness often helps model designers to be structured and hence the modeling exercise is less prone to modeling errors. The higher expressiveness accompanied by full power of a standard programming language such as Java/C++, tends to lead users to be too creative in their model building exercise which may be a source of many errors. It is our opinion that low expressiveness/high fidelity is the desired quality of a good modeling and simulation framework.

The term *low expressiveness* may mislead readers to think that such frameworks are not capable of achieving much. But since we pair *high fidelity* with low expressiveness, we insist that such frameworks can provide structures for modeling most common models of computation needed for an application domain. For example, Ptolemy II (if restricted only to use the existing MoCs, and not full expressive power of Java) is a low expressive/high fidelity framework for embedded software. However, ForSyDe in our view is low expressive/high fidelity for creating models with the purpose of abstraction and formal verification, but not necessarily for hardware/software co-design.

SystemC, as it is implemented in its reference implementation has a very low fidelity, because the only MoC that can be directly mapped to the current SystemC framework is the discrete-event MoC. The other MoCs needs to be indirectly mapped onto this particular

MoC whose semantics is suited for digital hardware simulation, and it is based very much on VHDL simulation semantics. However, if SystemC is extended with specific MoC structures and simulation facilities, it has the possibility of becoming a very high fidelity modeling framework for simulation, and synthesis of embedded hardware/software systems.

Another important metric to compare different frameworks is the simulation efficiency of complex models in the framework. It turns out that this metric is again closely tied to the fidelity and expressiveness issue. If a framework is structured to support multi-MoC modeling and simulation, it can be appropriately curtailed for simulation efficiency by exploiting the MoC specific characteristics. For example, consider the current reference implementation of SystemC with single discrete-event MoC kernel. Whenever an SDF model is simulated on such a kernel, the static scheduling possibility of SDF models goes unexploited and hence simulation efficiency is much less, compared to an extended framework based on SystemC, where SDF specific structures, and simulation kernels are available. Therefore, fidelity plays an important role in determining the efficiency of simulation of models in a framework.

3.1.1 Synchronous Data Flow MoC (SDF)

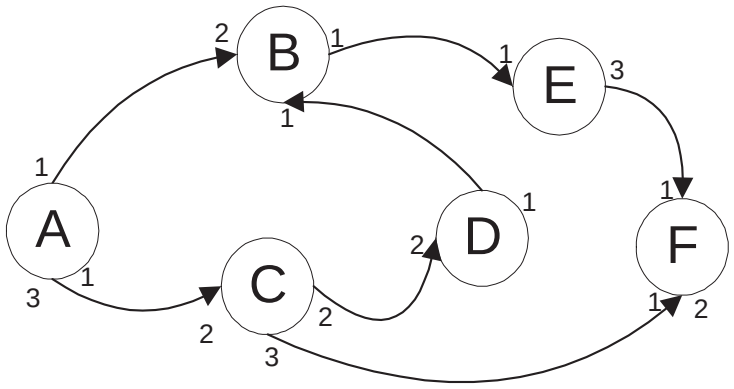


Figure 3.1: Abstract SDF Model Example from [33]

The synchronous data flow 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 3.1 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 [70, 33]. To compute the *executable schedule*, which determines which order and how many times each of the SDF blocks are to be fired, the *repetition vector* and the *firing schedule* need to be computed. The repetition vector simply indicates the number of times each of the SDF blocks must be triggered, whereas the firing schedule orders the SDF blocks in the correct order of execution for the specified number of times by the repetition vector [71, 33, 32]. One possible static executable schedule for the abstract SDF graph shown in Figure 3.1 is $2A1B1C1D1E3F$.

3.1.2 Finite State Machines MoC (FSMs)

Many researchers investigated the use of FSMs with concurrency models such as the co-design FSMs [72], SDF with FSMs [73], etc. Unfortunately, the problem here is that the concurrency model is highly coupled with the FSM semantics making it application specific. One of the most well known contributions to concurrent FSMs is Harel's Statecharts [74]. In Statecharts, a single state that contains a hierarchically embedded FSM can represent one of its hierarchically embedded states, commonly referred as OR states. Harel also describes the composition of FSMs referred to as AND states. However, ample Statechart variants proliferated after Harel's initial presentation of Statecharts due to the lack of a strict execution definition. The execution definitions describe when transitions are to be traversed, when hierarchical embeddings are to be triggered, and so on. For this reason, [75] emphasizes the importance of separating the concurrency MoC from the FSM semantics and develop the Starcharts (*charts) semantics. The main contribution of *charts is in decoupling the concurrency MoC from the FSM semantics with the understanding that a concurrency model can be combined with the *charts FSM to express the desired behavior. *charts show how to embed HFSMs and concurrency MoCs at any level in the hierarchy. The hierarchy can be of arbitrary depth and there are additional semantics that describe how it interacts with other MoCs. Starcharts 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. 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.

3.1.3 Communicating Sequential Processes (CSP)

In a CSP model, every node or block is a thread-like process that continuously executes unless suspended due to communication. These processes execute concurrently and rendezvous

communication protocol dictates that transfer of data only occurs when both communicating processes are ready to communicate [76]. If either of the processes is not ready to communicate then it is suspended until its corresponding process is ready to communicate, at which it is resumed for the transfer of data.

We further elaborate this rendezvous protocol by describing Figure 3.2. T1 and T2 are threads that communicate through the CSP channel that we call C1. T1 and T2 are both runnable and have no specific order in which they are executed. Let us consider process point 1, where T1 attempts to put a value on the channel C1. However, process T2 is not ready to communicate, causing T1 to suspend when the *put(...)* function within T1 is invoked. When process T2 reaches point 2 where it invokes the *get(...)* function to receive data from C1, T1 is resumed and data is transferred. In this case T2 receives the data once T1 resumes its execution. Similarly, once T2 reaches its second invocation of *get(...)* it suspends itself since T1 is not ready to communicate. When T1 reaches its invocation of *put(...)*, the rendezvous is established and communication proceeds. CSP channels used to transfer data are unidirectional. That means if the channel is going from T1 to T2, then T1 can only invoke *put(...)* on the channel and T2 can only invoke *get(...)* on the same channel.

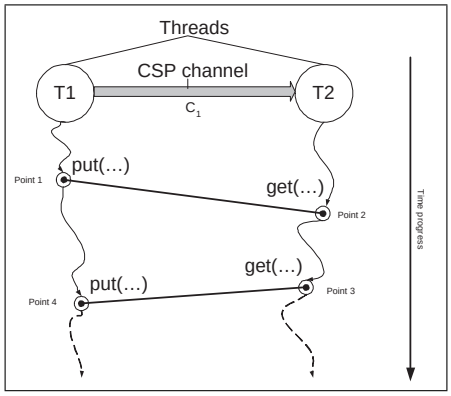
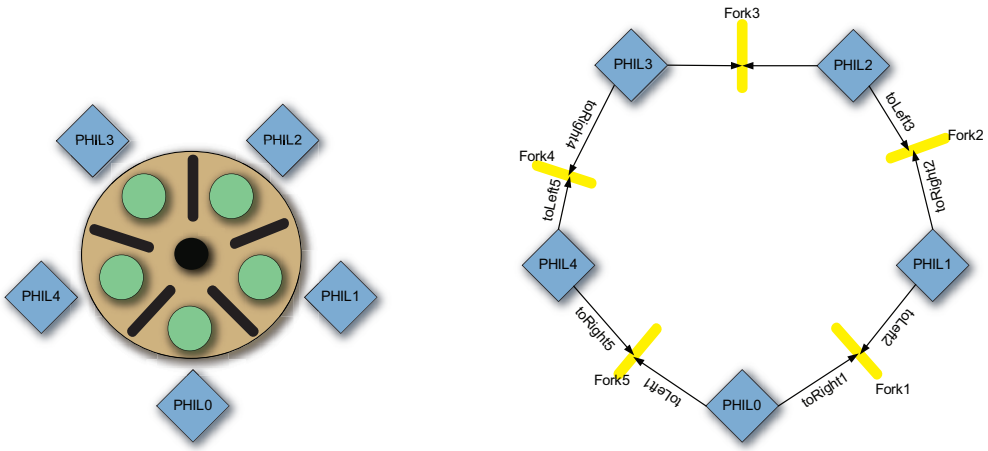


Figure 3.2: CSP Rendezvous Communication

To illustrate the *Communicating Sequential Processes* MoC, we take the classical *Dining Philosopher's* problem cited in [76]. The problem is defined as follows: there are five philosophers PHIL0, PHIL1, PHIL2, PHIL3, and PHIL4, there is one dining room in which all of them can eat, and each one has their own separate room in which they spend most of their time thinking. In the dining room there is a round table with five chairs assigned to each individual philosopher and five forks down on the table. In the middle of the table is a big spaghetti bowl that is replenished continuously (so there is no shortage of spaghetti).

The duty of the philosophers is to spend most of their time thinking, but when hungry they are to enter the dining room, sit on their designated chair, pick up their left fork then pick up their right fork and begin eating. Since a philosopher requires two forks to eat the spaghetti, another philosopher requesting for a fork that is unavailable must simply wait until it is put



(a) Block View (b) Implementation Schematic

Figure 3.3: Dining Philosopher

down. Once done eating, the right fork is dropped, then the left and then the philosophers go back to their respective rooms to think. This classical example is tackled using threading and mutexes many times.

Figure 3.3(a) displays the seating assignments of the philosophers. It also shows the dining room with the spaghetti bowl, the forks, and the assigned chairs. Though it may seem awkward to propose that inanimate objects have behavior, the forks in the Dining Philosopher model are processes along with the philosophers themselves. Therefore, there are a total of ten processes. The forks have distinct behaviors or realizing whether they are down on the table or whether they are not and the philosophers exhibit the behavior of thinking, picking up the left fork followed by the right, eating, then standing up to leave to think again.

Dining Philosopher Example in CSP Notation

In order to provide clarity to this example, we present the *Dining Philosophers* solution in the CSP [77] notation. CSP is represented as a process algebra, and hence each process is a term in the algebra. Here we informally describe the operators of CSP and refer the readers to [77] for a detailed syntax and semantics of CSP.

In CSP, a sequential process can be described with a sequencing operator. This operator prefixes an *atomic action* a to a process term b. The new process constructed represents a process that engages in action a first, and then like process P. For example

$$Q = a \rightarrow P$$

means that process Q executes action a and then behaves like a process represented by the process term P.

One can also represent *non-deterministic choice* between two behaviors with a *choice* operator written as |. For example

$$Q = a \rightarrow P \mid b \rightarrow R$$

is used to represent a process Q, which behaves in two different ways based on which atomic action it can engage in, a or b. If it can engage in a first, then it executes action a first, and then behave like the process represented by the process term P, else it engages in b and then behaves like the process represented by the process term R. If both a and b are possible actions that it can engage in, then it non-deterministically chooses one of the two behavioral courses. One can naturally extend the choice operator from binary to n-ary, so a process can have more than two non-deterministic choices and courses of actions. Often times if the action set is indexed, for example if $a[i]$ such that $i = 0..n$, and the behaviors following action $a[i]$ is describable by a process term also indexed by i, say $P[i]$, then a compact representation of the choice is written as

$$Q = |_{i=0..n} \ (a[i] \rightarrow P[i])$$

Although the original CSP has no direct syntax to describe *priority* among the possible choices, later on variants of CSP have been proposed to encode priority among the choices. Priority can be partially ordered in general, but for our purposes we assume total order on the list of choices in terms of priority. So we will use a $\rightarrow$ on top of the choice operator |, and write it as $\overrightarrow{|}$ to denote prioritized choice with the direction of priority being decreasing from left to right in the term description.

To represent indefinite repetition of a sequential behavior, often written with loops in standard programming notation, CSP allows use of recursive definitions of process terms. For example

$$Q = a \rightarrow Q$$

is used to represent a process which indefinitely repeats the action a. One can have a combination of choice and recursion to encode a process which goes on engaging in a sequence of actions until a certain action happens, when it can change its behavior. For example,

$$P = a \rightarrow Q \mid b \rightarrow P$$

represents a process that can either engage in an indefinitely long sequence of action b, or if it can engage in action a, then it changes its behavior according to process described by the term Q.

Given that we are now able to represent sequential processes with the above described notations, we can describe *parallel composition* with *rendezvous* synchronizations. The parallel operator is denoted by ||, and the synchronization alphabet accompanying the parallel operator notation. However, to avoid complicated notation, we will specifically mention the synchronization actions separately rather than burdening the parallel composition operator. Let us assume that process P engages in actions in the set $\alpha(P) = \{a, b, c\}$, and process Q

engages in actions in the set $\alpha(Q) = \{a, b, d, e\}$, then the parallelly composed process $P \,||\, Q$ must synchronize on the action set $\alpha(P) \cap \alpha(Q) = \{a, b\}$. This means that whenever process P will come to a point when it is ready to engage in action a, process Q must be ready to engage in action a, and they would execute action a together. If Q is not ready for engaging in action a, at the time P is ready to do so, P must get suspended until Q executes further and gets ready to engage in this synchronization action. This is what is known as *rendezvous* synchronization. For example,

$$P = a \rightarrow c \rightarrow b \rightarrow P,$$

and

$$Q = d \rightarrow a \rightarrow e \rightarrow d \rightarrow b \rightarrow Q,$$

when parallely composed into $P \,||\, Q$, then P has to remain suspended until Q finishes its independent action d, and then they can execute the synchronous action a together, after which P can do its independent action c while Q does its sequence of e and d, before the two of them synchronize again on b.

Given this background, we are ready to describe the *Dining Philosopher* solution based on [77], which are attributed to E. W. Dijkstra and C. Scholten.

As per the figure shown, we have 5 philosopher CSP threads, and 5 fork CSP threads. The philosopher processes are called $PHIL_i$ where $i = 0..4$, and the fork processes are denoted as $FORK_i$ where $i = 0..4$. Due to the symmetric nature of the processes, we will describe the ith process only. The action alphabet of $PHIL_i$ is given as

$$\begin{aligned}\alpha(PHIL_i) = \{&think_i, requestseat_i, getfork^i_{i-1}, getfork^i_{i+1}, eat_i,\\ &dropfork^i_{i-1}, dropfork^i_{i+1}, relinquishseat_i\}.\end{aligned}$$

Similarly for the fork processes

$$\begin{aligned}\alpha(FORK_j) = \{&getfork^{j-1}_j, getfork^{j+1}_j, dropfork^{j-1}_j,\\ &dropfork^{j+1}_j\}.\end{aligned}$$

Note that $i-1, j-1, i+1, j-1$ are modulo 5 increment and decrement operations. Which means when $i = 4, i+1 = 0$, similarly, when $i = 0, i-1 = 4$ etc. Now we can write down the CSP process describing the behavior of the philosopher number i as follows:

$$\begin{aligned}PHIL_i = &think_i \rightarrow requestseat_i \rightarrow getfork^i_{i-1} \rightarrow getfork^i_{i+1} \rightarrow eat_i\\ &\rightarrow dropfork^i_{i-1} \rightarrow dropfork^i_{i+1} \rightarrow relinquishseat_i \rightarrow PHIL_i.\end{aligned}$$

Similarly, behavior of the process describing the fork number j is given as

$$\begin{aligned}FORK_j = getfork^{j-1}_j \rightarrow dropfork^{j-1}_j \rightarrow FORK_j \overset{\rightarrow}{|} getfork^{j+1}_j \rightarrow\\ dropfork^{j+1}_j \rightarrow FORK_j\end{aligned}$$

Note that in this particular solution, $FORK_j$ prioritizes between the philosopher at its left and the one at its right, so that the solution is tenable to the case when the number of philosophers is even as well.

So the dining philosopher system can now be written as

$$DP = ||_{i=0}^{i=4} (PHIL_i \,||\, FORK_i)$$

where $||_{i=0}^{i=4}$ is an obvious indexed form of the parallel composition operator.

Unfortunately as explained in [77], this solution can deadlock when all the philosophers decide to ask for the fork on their left at the same time. This necessitates an arbitrator, and a solution is given in the form of a footman, invented by C. S. Scholten. In our examples, we use this footman based solution to illustrate a finite state machine control of the dining philosopher system. Here we describe in CSP notation, how the footman works. Basically, the footman is the one who receives request for a seat by a hungry philosopher, and if the number of philosophers eating at that time is less than four, only then it grants such a request. This avoids the deadlock scenario described above. We describe the CSP notational form of the footman following [77] with a mutually recursive definition. Let us denote the footman as $FOOT_n^S$ for $n = 0..4$, such that $FOOT_n^S$ denotes the behavior of the footman when n philosophers are seated and the set S contains the indices of those who are seated. The alphabet of $FOOT_n^S$ is given by $\cup_{i=0}^{i=4}\{requestseat_i, relinquishseat_i\}$, and hence those are the actions of $PHIL_i$ which needs to synchronize with the footman. Now we describe the CSP terms for the $FOOT_n^S$ as follows:

$$\begin{aligned}
FOOT_0^{\{\}} &= |_{i=0}^{i=4} (requestseat_i \rightarrow FOOT_1^{\{i\}}) \\
FOOT_1^{\{i\}} &= (|_{j \neq i} (requestseat_j \rightarrow FOOT_2^{\{i,j\}})) \\
&\quad | (relinquish_i \rightarrow FOOT_0) \\
FOOT_2^{\{i,j\}} &= (|_{k \neq i,j} (requestseat_k \rightarrow FOOT_3^{\{i,j,k\}})) \\
&\quad | (|_{l \in \{i,j\}} (relinquish_l \rightarrow FOOT_1^{\{i,j\}-\{l\}})) \\
FOOT_3^{\{i,j,k\}} &= (|_{l \neq i,j,k} (requestseat_l \rightarrow FOOT_4^{\{i,j,k,l\}})) \\
&\quad | (|_{x \in \{i,j,k\}} (relinquish_x \rightarrow FOOT_2^{\{i,j,k\}-\{x\}})) \\
FOOT_4^{\{i,j,k,l\}} &= |_{x \in \{i,j,k,l\}} (relinquish_x \rightarrow FOOT_3^{\{i,j,k,l\}-\{x\}})
\end{aligned}$$

So with the footman the dining philosopher system is now described as

$$DP = FOOT_0^{\{\}} \,||\, (||_{i=0}^{i=4} (PHIL_i \,||\, FORK_i))$$

We will show in later chapters that each $PHIL_i$ can contain computations which are more involved rather than being just atomic actions as shown here. In fact, the computations involved may actually require computation in another MoC. The footman can be also implemented in FSM MoC, rather than keeping it a CSP process.

3.1.4 Abstract State Machines (ASMs)

Simply states, Abstract State Machines (ASMs) [78, 79] are finite sets of transition rules. A transition rule consists of a guard and an action. A transition rule looks like

if Guard then Updates

where the "Guard" evaluates to a Boolean value and Updates is a finite set of assignments. The set of assignments update values of variables of the state. These assignments are depicted as

$$f(t_1, ..., t_n) := t_0$$

where t_0 to t_n are parameters and f denotes a function or a variable (a 0-ary function). At each given state (also referred to as a step), the parameters t_0 to t_n are evaluated first to obtain their values denoted by v_k for $k = \{0, ..., n\}$ respectively. Upon the evaluation of v_k, the functions in the Updates set are evaluated. Hence, $f(v_1, ..., v_n)$ is updated to v_0. A run of an ASM simultaneously updates all transition rules whose guard evaluates to true in that state.

There are several ASM variants whose basis is the standard ASM as described above. Examples of the variants are Turbo-ASMs [78], synchronous ASMs and distributed/asynchronous ASMs [78, 47]. Each of these variants posses certain descriptive capabilities. For example, Turbo-ASMs are appropriate for expressing sequential composition, *while* loops and other specific constructs aiding in composability.

3.1.5 SpecExplorer

SpecExplorer is a specification exploration environment developed my Microsoft [80, 81] that is used for model-based specification and conformance testing of software systems. The language used for specification can be either AsmL [82, 83], a derivative of ASMs or Spec# [84, 85].

Exploration in SpecExplorer generates automata from the specification. This requires indicating the actions and states of interest. The four types of actions supported are controllable, observable, scenario and probe. Controllable actions are those that the specification invokes and the observable are the ones invoked by the environment. A scenario action brings the state elements to a particular starting state and the probe action is invoked at every state in efforts to look into the implementation. There are a number of exploration techniques that can be used to prune the state space. For example, state filters are Boolean expressions that must be satisfied by all explored states. Similarly, the state space can be limited to a set of representative state elements as well as state groupings. Finally there is support for generating test cases from the automaton generated from the exploration. SpecExplorer supports coverage, shortest path and random walk test suite generation. These are discussed in further detail in the reference documents of SpecExplorer [80].

3.1.6 Discrete-Event Semantics Using ASMs

Our Discrete-Event semantics specification uses Turbo-ASMs with object-orientation and polymorphism capabilities of AsmL. We specify the simulation semantics in the `DiscreteEvent` class shown in Figure 7.1. The numbers associated with each function shows the order in which these are executed. For example, after the instantiations of the variables in (1), the entry point is the `triggerDiscreteEvent` function marked with (2). Note that we use AsmL's parallel constructs such as the `forall` in `triggerBehaviors` to specify parallel execution of the behaviors. This corresponds well *to the unspecified execution order of the processes* in SystemC's simulation semantics. We have created additional classes that help the designer in describing a semantic model in AsmL so that it follows the our Discrete-Event simulation semantics. Examples are the `deNode` class that represents the behavior of a specific process and the `deGraph` that represents the netlist of the entire design. Anyhow, we do not present these here because the focus here is test generation. In the future, we will make the entire semantics downloadable via the web at [86].

The simulation begins with a function `start` that is not shown in Figure 7.1. This function updates the state variables `stopTime` and `designGraph` that hold the duration of the simulation and a structural graph of the system being modeled respectively. After the values of these two variables are updated, the simulation begins by invoking `triggerDiscreteEvent`. This initializes the simulation by triggering every behavior in the system that in turn generate events. After this, the simulation iterates through the `evaluate`, `update` and `proceedTime` functions. In AsmL `until fixpoint` only terminates when there are no updates available or there is an inconsistency in which the simulation throws an exception. The `processEvents` function retrieves a set of events that match `simClock` and these events are triggered via `triggerBehaviors`. The `update` function invokes an update on all specified channels very much like SystemC and the `proceedTime` forwards the simulation time by calling `nextEvent`. The `nextEvent` returns an event from the event set. The semantics of this enforce the processing of all delta events first before a timed event.

3.1.7 Exploration in SpecExplorer

SpecExplorer provides methods for generating automata based on the designer's input for exploring the specification. The designer specifies the actions of interest in the exploration settings, for an FSM to be generated. These actions are functions in the specification. They can be classified into four types: controllable, observable, scenario and probe [80]. Controllable typed actions are the functions that are invoked by SpecExplorer and observable are the actions that SpecExplorer waits for a response from the implementation model. Probe actions simply query for state information and scenario actions provide a way of reaching a starting state for the exploration. Selectively exploring transitions of the specification is possible via a variety of methods such as parameter selection, method restriction, state filtering and state groupings. We direct the reader to [80, 87] for further information regarding

exploration techniques in SpecExplorer.

Accepting states describe the states at which a test must finish. These accepting states are defined using a state-based expression. For example, a simple `true` suggests that all states are accepting states and `FULL = true` suggests that only the states where the state variable `FULL` is true are accepting states. The test case generator uses this state-based expression and computes all possible paths from the initial state to the accepting states given that all other constraints such as state filters are satisfied. Our methodology uses the accepting states and methods for selectively exploring transitions for directing the test case generation and diagnosis.

Chapter 4

Behavioral Hierarchy with Hierarchical FSMs (HFSMs)

We introduce the idea of behavioral hierarchy in this chapter using examples to illustrate the difference between traditional modeling and simulation and one that preserves the behavioral hierarchy. However, an intuitive understanding of behavioral hierarchy is insufficient in being able to implement it such that it is suitable for extension for heterogeneous behavioral hierarchy, which is another essential ingredient for SLDLS. We present the basic formal definitions allowing us to formalize the execution semantics for the hierarchical FSM MoC. This sets the foundation for extending the formalization of the HFSM with heterogeneity that we will see in the upcoming chapters.

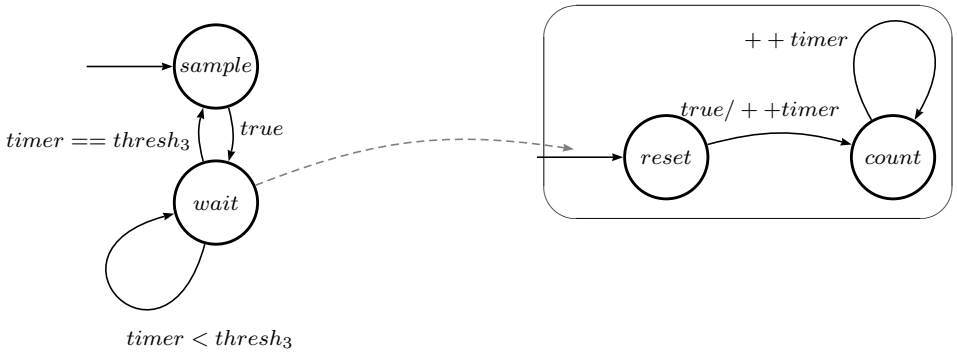


Figure 4.1: Small Segment of the Power State Machine Model

Using the sampler component from the power state machine model depicted in Figure 4, we provide an intuitive understanding of *behavioral hierarchy*. The sampler component's behavior is decomposed into one that samples for a task and one that waits $thresh_3$ units of time before every sampling. We represent these two behaviors as FSMs, where Figure 4

shows a two-state FSM with states `sample` and `wait`, with an embedded FSM within the `wait` state representing the timer. From this small example, we see that the advantages of using behavioral decomposition are the ease and clarity of modeling and the reuse of already modeled behaviors in different parts of the model. For example, we use three instances of the FSM timer from Figure 4 in our power model shown in Figure 5.7. In addition, when a simulation framework takes advantage of this hierarchical embeddings, it may result in better simulation performance. Thus, *behavioral hierarchy* is the preservation of the hierarchy information during modeling and during simulation.

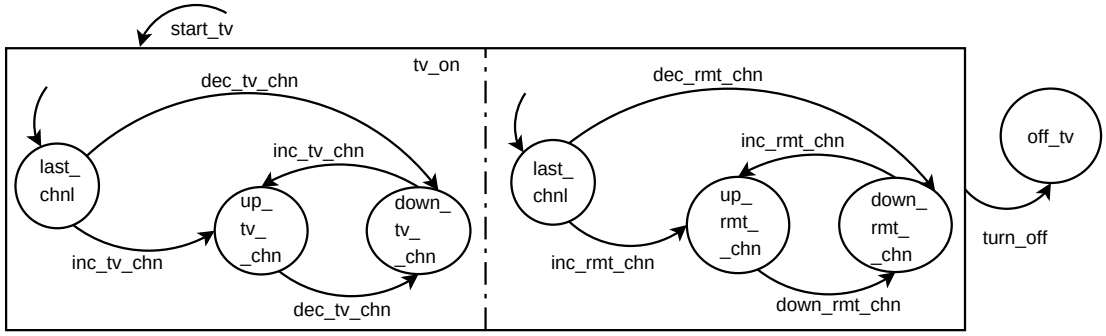


Figure 4.2: Hierarchical FSM

Another advantage of using behavioral hierarchy is abstraction of state space. To understand how behavioral hierarchy helps in the state space abstraction, let us look at a simple television control example in Figure 4.2 and Figure 4.3. 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_`). This example also shows AND concurrency in FSMs [74]. In Figure 4.2, the `tv_on` state encapsulates all behaviors of changing the channels via the remote or the television. Notice that only one transition to `off_tv` is required from this state, whereas in Figure 4.3 every state must have a transition to turn the television off. Furthermore, the toplevel FSM in Figure 4.2 only sees two states, `tv_on` and `off_tv` and the hierarchical FSM sees the six other states. Figure 4.2 can use three instances of an FSM scheduler to simulate the model. One instance simulates the two toplevel states and the `tv_on` can use one instance for the remote controls and the other for the television controls. Hence, abstracting the state space effectively.

This only shows an example of HFSMs, but behaviors can be decomposed into heterogeneous MoCs suggesting that behavioral hierarchy can also be performed on heterogeneous components. Figure 4.4(a) shows a block diagram of a design that employs behavioral hierarchy with heterogeneous components, which we term *heterogeneous behavioral hierarchy*. The ability to analyze a design by dissecting it into small behaviors and then composing these behaviors in a hierarchy is what we term *behavioral hierarchy*, given that the same MoC is used for its realization. The separation between behavioral hierarchy and structural hierarchy enables designers with the capability to express behaviors from small and simpler behaviors to construct behavioral models in a hierarchical fashion and because behavioral

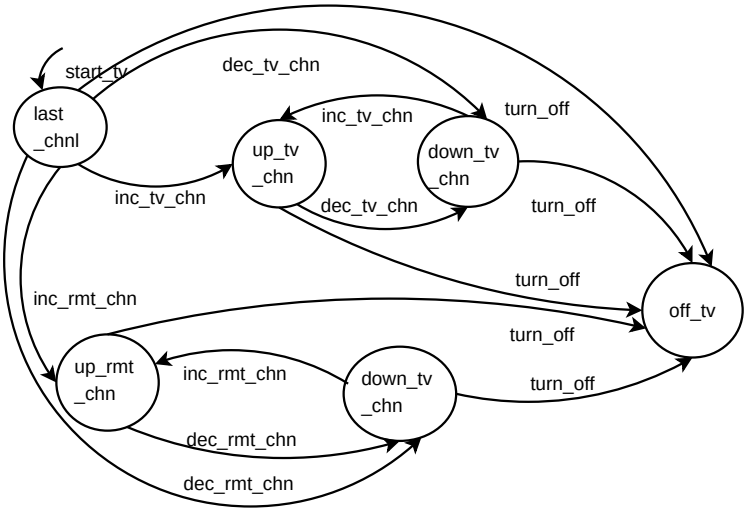
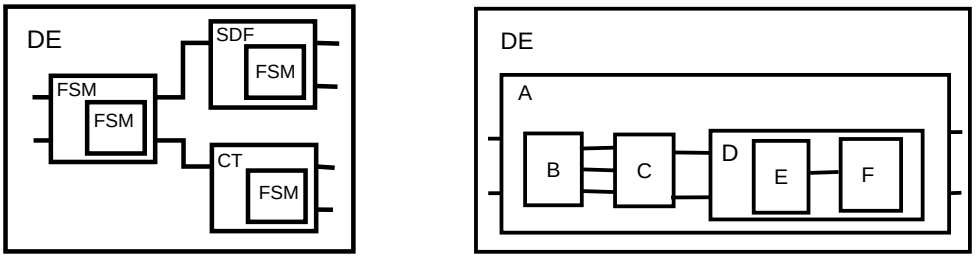


Figure 4.3: Equivalent Flat FSM

hierarchy preserves the behaviors at every level of the hierarchy, during simulation this extra hierarchy information can be used. Support for behavioral hierarchy in simulation suggests exploitation of the hierarchical structure in the behavioral model to increase simulation speed and maybe improve synthesis results.



(a) Heterogeneous Behavioral Hierarchy

(b) Structural Hierarchy

Figure 4.4: Examples of Heterogeneity and Hierarchy

4.1 Behavioral Modeling versus Structural Modeling

SLDLs such as VHDL and Verilog both have behavioral modeling but lack in behavioral hierarchy modeling. Behavioral modeling refers to decomposing designs based on their behaviors such that the resulting design describes a sequence, composition and hierarchy of behaviors. On the other hand, structural modeling also uses behavioral decomposition, except that it is only performed at the RTL abstraction level. This means that behaviors are modeled in terms of registers, AND/OR/NOT gates, wires, logic expressions and assignments. However,

once a behavior is defined, it is then composed structurally. For example, a four-bit adder may consist of four one-bit adders where at first, a one-bit adder is modeled at the RTL abstraction level, after which four instances of one-bit adders are connected to represent a four-bit adder with additional glue logic. This is what we call *structural modeling*. Now, when these components are composed to form a hierarchy, they exercise *structural hierarchy*. Figure 4.4(b) shows a block diagram where the components A-F are composed exhibiting structural hierarchy.

By using structural hierarchy, we loose out on the two main advantages obtained from using behavioral hierarchy. The first disadvantage is that simple behaviors 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 model construction, the SystemC simulation scheduler flattens the structural hierarchy not taking advantage of the state space abstraction through hierarchy. In the context of SystemC, if each state is modeled as a SystemC process and transitions as signals, the flat FSM in Figure 4.3 requires six SystemC processes. Similarly, for Figure 4.2, SystemC expands the hierarchy and realizes the model as eight process where the `tv_on` is also a SystemC process. This shows that structural hierarchy clearly 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. The solution we propose is behavioral hierarchy.

4.2 Finite State Machine Terminology

FSMs are commonly represented as directed graphs where the nodes represent the states and the arcs represent the transitions, similar to state transition diagrams. There are various HFSM formalisms but the one we use is that of the Ptolemy II group's called Starcharts or **charts*. Every transition has a guard/action pair where the guard must be a Boolean expression using the inputs and the actions represent the outputs to be present when the guard evaluates to true. 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. 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.

Suppose we have a simple HFSM as shown in Figure 4 where the `wait` state is refined as another FSM. The master FSM consists of `sample` and `wait` states, with the slave subsystem containing the states `reset` and `count`. The initial state for the master FSM is `sample` and for the slave subsystem it is `reset`, indicated by the incoming edge without a source state. Transitions that do not appear to have an action associated with the guard assume that

there is no action or that state change for that particular transition. According to *charts semantics, when the master FSM executes, the current state `sample`, transitions to the `wait` state, which completes one reaction of the FSM causing it to return. Another firing of the master FSM requests one reaction of the master FSM again. However, during the second reaction, since `wait` is a refined state, its refinement has to react first. This means the slave subsystem executes by entering the `reset` state and transitions to the `count` state. The timer value is updated and since the slave system has successfully reacted once, it returns and allows the reaction of the master FSM to complete. This involves checking whether any of the transitions from the `wait` state are enabled and if not then it returns. Details of *charts are available at [75].

4.3 Requirements for Behavioral Hierarchy in SystemC

First, we enlist some of the general requirements to enable SystemC for behavioral hierarchy followed by specific requirements to enable the hierarchical FSM MoC in SystemC. The general requirements apply to all extensions of SystemC. For example, we noticed when implementing the SDF, CSP and FSM MoCs for SystemC-H [26] that we require a uniform method of describing the structure of the design. An appropriate representation is in the form of a directed graph, thus making a generic graph representation a general requirement. We summarize these general requirements as follows:

- The existing SystemC standard remains unaffected, thereby allowing existing SystemC models to execute without alterations.
- None of the SystemC source is altered.
- The extensions must be packaged separately such that integration does not induce any changes to the reference implementation.
- A generic graph representation to describe the designs created using the extensions.

4.3.1 Requirements for Hierarchical FSMs

The foremost requirement in extending SystemC with HFSMs was representing the structure of HFSMs. That means we require a way to describe the states, transitions, the arcs or edges that connect states with transitions and so on.

- A designer specifies HFSMs in the following manner

  - State objects are connected via transition objects to other state objects. The order in which they are connected enforces a relation that describes from which state to which state the transition is directed.
  - Transition objects constitute of a guard and action pair.
  - Choice actions associated with a transition object are triggered every time the transition is enabled.
  - Commit actions associated with a transition object are only triggered before the change of state for the HFSM.
  - Member functions are added to set properties of states and transition objects.

- Upon entering a state, the entry actions of that state are triggered.
- Just before changing state, if the current state has exit actions then those are triggered.
- The run-to-complete property for transition and state objects ensures the refinements in the transition traverse from initial to final state before returning control. However, an object cannot contain orthogonal HFSMs as refinements and also be set for run-to-complete.
- A preemptive transition executes the choice actions whenever it is enabled and skips execution of the refinement of the current state..
- A reset transition resets the refinement the refinement of the destination state after performing the commit actions for that transition object.

4.3.2 Additional Semantic Requirement for Hierarchical FSMs

As mentioned earlier, there are many variants of Harel's Statecharts that intertwine concurrency MoCs with Statecharts to increase their expressiveness. However, Ptolemy group's philosophy is to decouple the concurrency model from the semantics of HFSMs and hence the *charts execution semantics. Though this decoupling provides a clean semantic foundation for multi-MoC designs, it requires the introduction of multiple concurrency models for the FSM MoC to be of use in system designs. Sometimes, source code alterations are required to perform some intuitive tasks for HFSMs. For example, a run-to-complete (not to be confused with the Run To Completion (RTC) in UML Statecharts) FSM refinement would require adding functionality to the Ptolemy II's FSMDirector or using several other higher-order actors to provide a work-around. We decided that it is important for designers to specify whether a particular FSM must only return once it traverses from its initial to final state, hence the run-to-complete. The *charts execution semantics does not easily allow designers to implement this run-to-complete, which we see as important. Therefore, we implement it directly as a feature of the HFSM extension. The basic intuition behind run-to-complete is that the refinement must traverse from its initial to final state and only

returns control to the master kernel once the final state is reached. Both transitions and states can be set run-to-complete. However, this is disallowed when there are orthogonal FSMs as refinements. We also include the idea of entry and exit actions for states [74]. Entry actions are performed as soon as the HFSM enters that state, whereas the exit actions are performed just before the change of state of the HFSM.

4.4 Execution Semantics for Hierarchical FSMs

This section begins by presenting our abstract semantics that every executable entity implements. This is followed with formalized definitions used in describing the execution semantics we implement in our HFSM MoC. In later chapters we use the same definitions and redefine them for supporting heterogeneous behavioral hierarchy.

4.4.1 Abstract Semantics

Listing 4.1: Executable Entity interface

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

Our implementation follows an abstract semantics similar to Ptolemy II's `prefire()`, `fire()`, `postfire()`. Our approach declares `pre_prepare()`, `prepare()`, `precondition()`, `execute()`, `postcondition()` and `cleanup()` as shown in Listing 5.1. Each of these pure virtual member functions have a specific responsibility. The `pre_prepare()` is a step before the initialization of the system and 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 `pre_prepare()`, `prepare()` and `cleanup()` member functions are only invoked once during initialization and then termination of the executable entity. For example, the `prepare()` of the `fsm_model` is responsible for ensuring that the initial state is defined and that the initial state is not the same as the final state for that particular FSM. Note in Listing 5.1 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. This includes every executable entity in all the MoCs.

It is important to understand the simplicity of this approach and the benefit achieved when introducing heterogeneous behavioral hierarchy. Take 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 as it will be described in later chapters.

4.4.2 Basic Definitions for HFSMs

The basic definitions constitute of defining the sets for FSM states and transitions, functions to check if an object has a refinement, to distinguish the type of the object, whether the run-to-complete property of an object is set and to iterate through all refinements of an object. Please note that we redefine these basic definitions in later chapters to extend them for heterogeneous behavioral hierarchy.

Definition 4.4.1.
Let ST be the set of state objects
Let TR be the set of transition objects
Let $FSMST$ be the type of state objects
Let $FSMTR$ be the type of transition objects

Definition 4.4.2. *To distinguish the type of an element.*

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

The $\tau()$ function takes in either a state or transition object and returns whether it is a state or transition object.

Definition 4.4.3. *Identify if entity has refinement.*

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

The HFSM MoC allows embedding of other HFSMs within either state or transition objects, which are refinements of the component embedding them. The $\rho()$ function checks whether there is a refinement within either a state or transition object. Since we simulate one level of hierarchy at a time, we only need to check for a state or transition having at the very least one refinement. Further hierarchical refinements are handled by their respective instances of the kernel.

Definition 4.4.4. *Check if object is set to run-to-complete*

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

This run-to-complete feature is not available in *charts. However, we deem it important and a useful feature for hardware modeling; hence we incorporate it into our implementation of the HFSMs. The basic intuition behind run-to-complete is that the refinement must traverse from its initial to final state and only returns control to the master kernel once the final state is reached. Similar to the property of transitions being set run-to-complete, states also can have refinements set to run-to-complete. This means that every refinement one level of hierarchy deeper must adhere to the run-to-complete requirements.

Definition 4.4.5. ***trigger*** *– Iterate through all refinements for a particular object.*
Let R be the set of refinements for a particular object
Let $get\_all\_refs(x)$ return all refinements for object x where $x \in \{ST \cup TR\}$
Let $iterate(x)$ perform an iteration on the object x

$$\begin{aligned} trigger(x) = \quad & \textbf{\textit{Require:}}\ R = \{\emptyset\} \\ & R = get\_all\_refs(x) \\ & \forall\ r \in R\ \textbf{\textit{do}} \\ & \quad \underline{iterate(r)} \end{aligned}$$

The $trigger()$ function is used to iterate through all the refinements of a particular object. This is required because a particular object may have more than one refinement, which need to be iterated through.

4.4.3 Execution Semantics for Starcharts

We step through the *charts execution semantics using the algorithm presented in Algorithm 3. However, to first provide an intuitive understanding of hierarchical FSMs using

Algorithm 1 Algorithm for execution semantics for HFSMs

```
Let currST be the current state
α = get_enabled_tr(currST)
if α ≠ ∅ then
  if ρ(α) == 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}
α = get_last_enabled_tr()
if α ≠ ∅ then
  execute_commit_action(α)
  currST = next_state(α)
  if π(α) == RESET then
    reset_state(currST)
  end if
end if
```

*charts, let us take the simple example shown in Figure 4.2 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 3, the current state of the HFSM is *currST* and the *get_enabled_tr*() retrieves an enabled transition from *currST*, ensuring that there is a maximum of one enabled transition from a state to satisfy the determinism property of *charts. 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 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. 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. 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.

4.4.4 Our Execution Semantics for Hierarchical FSMs

Continuing to use the definitions presented earlier, we provide few additional definitions that allow us to present our execution semantics for hierarchical FSMs. Our algorithm is a slight variant of the *charts 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 its refinement's HFSM from its initial to final state. We also incorporate Harel's entry and exit actions for states [74] giving the states the responsibility of encapsulating the actions rather than simply labels to which transitions are attached.

Additional Definitions Specific for HFSMs

The $rtc\_iterate()$ function performs run-to-complete iterations on the object x. In the current definition, it invokes a function that is specific to the HFSM MoC, $rtc\_fsm\_iterate()$. This performs one iteration on either the state or transition object. As a reminder, one iteration is an execution of the `precondition()`, `execute()` and `postcondition()`.

Definition 4.4.6. *Run-to-complete iterate for different HFSMs.*

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

Definition 4.4.7. *Return refinements for particular object.*
Let $get\_state\_refs(x)$ return a set containing the refinements in the state.
Let $get\_transition\_refs(x)$ return a set containing the refinements in the transition.

$$get\_all\_ref(x) = \{\ Y \mid \forall x \in \{ST \cup TR\}, Y = f_y(x)\ \}$$
$$where\ f_y(x) = \begin{cases} get\_state\_refs(x) & if \tau(x) = FSMST \\ get\_transition\_refs(x) & if \tau(x) = FSMTR \end{cases}$$

We define $get\_all\_ref()$ that returns a set containing all the refinements in an object. The function $get\_state\_refs()$ is invoked if the object x is a state object and $get\_transition\_refs()$ is invoked if the object is a transition object. The resulting set Y has the refinements embedded in that object x.

Definition 4.4.8. *To distinguish the type of transition objects.*

$$\pi(x) = \begin{cases} PREEMPTIVE & x \text{ is set to preemptive} \\ NONPREEMPTIVE & x \text{ is set to nonpreemptive or default} \\ RESET & x \text{ is set to reset} \end{cases}$$
$$where\ x \in TR$$

The $\pi()$ function determines the different types of possible transitions. The four types of transitions it recognizes are *PREEMPTIVE*, *NONPREEMPTIVE*, *RESET* and *RUN-TO-COMPLETE*. 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.

Definition 4.4.9. ***trigger*** – *Iterate through all refinements for particular object.*
Let R be the set of refinements for particular object
Let $iterate(x)$ perform an iteration on object x

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

Lastly, we redefine the $trigger()$ procedure to incorporate run-to-complete iterations into the execution semantics. Notice the difference between the redefinition of $trigger()$ in Definition 4.4.9 and Definition 4.4.5. The only change we make is in checking for run-to-complete objects and using the $rtc\_iterate()$ function instead of the $iterate()$ function..

We accommodate the run-to-complete semantics into the semantics of *charts in the previous definitions. However, we have yet to update the algorithm such that entry and exit actions

Algorithm 2 Algorithm for execution semantics for HFSMs with entry and exit actions

```
Let currST be the current state
execute_entry_state_actions(currST)
α = get_enabled_tr(currST)
if α ≠ ∅ then
  if ρ(α) == 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}
α = get_last_enabled_tr()
if α ≠ ∅ then
  execute_commit_action(α)
  execute_exit_state_actions(currST)
  currST = next_state(α)
  if π(α) == RESET then
    reset_state(currST)
  end if
end if
```

are also treated correctly. For this we present a slightly altered version of Algorithm 3 in Algorithm 2. The underlined function invocations show that the algorithm first handles the entry actions of the current state and before changing the current state to the next state, the exit actions are performed. This with the redefinition of $trigger()$ along with the additional functions completes our definition of HFSMs extension.

Now, we present the HFSM kernel's simulation algorithm as per making the kernel an executable entity. This entails giving the HFSM kernel a clear definition in terms of the abstract semantics we presented earlier; therefore basically separating Algorithm 2 into the respective implementations for the virtual members functions.

Definition 4.4.10. ***pre_prepare(...)*** *– Pre-prepare FSM objects in model.*

Require: $ST \neq \{\emptyset\}$, $TR \neq \{\emptyset\}$, $initST \neq \emptyset$, $finalST \in \{ST \cup \emptyset\}$, $currST = initST$

```
∀ α ∈ {ST ∪ TR} do
  pre_prepare(α)

return true
```

Definition 4.4.10 shows the *pre_prepare*() member function that simply ensures that there exist some states and transitions in the HFSM and that the initial state *initST* is defined and set to the current state *currST* of the HFSM. Due to the fact that either one of the states or transitions may have refinements, it is necessary to invoke *pre_prepare*() on every one of those objects. This will invoke *pre_prepare*() on every state or transition object and its refinements.

Definition 4.4.11. ***prepare(...)*** *– Prepare FSM objects in model.*

Require: $initST \neq finalST$
  $\forall\ \alpha \in \{ST \cup TR\}$ ***do***
    prepare(α)
  return *true*

The *prepare*() member function in Definition 1 checks that the final state is not the same as the initial state, because if this was the case then it is most likely a modeling error. Similar to the *pre_prepare*() member function, the *prepare*() invokes the *prepare*() on every transition and state along with any refinements and their refinements within it. Note the distinction that when a refinement's *pre_prepare*() or *prepare*() or any other member from the abstract semantics is invoked, it actually invokes it on the refinement's kernel. In our case the `fsm_director` instance associated with that level of hierarchy.

Definition 4.4.12. ***precondition(...)*** *– Precondition for FSM model.*

Require: $currST \neq \emptyset$
  $exec\_entry\_actions(currST)$
  return *true*

The *precondition*() in Definition 2 ensures that the current state is a valid state and then executes its entry actions. Since an HFSM is always ready to execute, we *true* after executing the entry actions.

Definition 4.4.13. ***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 $\alpha == \emptyset$ ***then***
    return *false*
  end if
  if $\pi(\alpha) == PREEMPTIVE$ ***then***
    if $\rho(\alpha) == true$ ***then***
      trigger(α)
    end if
    $execute\_choice\_actions(\alpha)$

return *true*
end if
$\underline{trigger(currST)}$
if $\pi(\alpha) == NONPREEMPTIVE$ ***then***
if $\rho(\alpha) == true$ ***then***
$\underline{trigger(\alpha)}$
end if
$execute\_choice\_actions(\alpha)$
end if
return *true*

The core of the functionality is implemented in *execute*() as shown in Definition 3. The first step is to retrieve an enabled transition using the *get_enabled_tr*() function. If there are more than one enabled transitions then this function returns no transitions. Therefore, a check on α will result in returning $false$. This is once again a modeling error since our semantics only support deterministic HFSMs and having more than one enabled transition implies non-determinism. We display an error message indicating the possible modeling error. Assuming that there is only one enabled transition, we check whether the transition type is $PREEMPTIVE$. If a $PREEMPTIVE$ transition has refinements, then the refinements are executed using the *trigger*() member, after which the choice actions associated with this transition are executed. This will conclude the *execute*() function if the enabled transition was $PREEMPTIVE$ by returning *true*. This means that the refinements of the current state are not triggered as per the requirement for $PREEMPTIVE$ transitions. However, if the enabled transition is not of type $PREEMPTIVE$ then we execute the current state's refinements by invoking $trigger(currST)$ and check for the transition being of type $NONPREEMPTIVE$. In a similar fashion to the $PREEMPTIVE$ transition, we *trigger*() the refinements if there are any and then execute the choice actions associated with this $NONPREEMPTIVE$ transition. Finally, we return *true* from the *execute*() function.

Definition 4.4.14. ***postcondition(...)*** – *Postcondition for FSM director.*
Let $lastTR$ *be the last enabled transition*
where $lastTR \in \{TR \cup \emptyset\}$
Require: $lastTR \neq \emptyset$, $currST \neq \emptyset$
$execute\_commit\_actions(lastTR)$
$exec\_exit\_actions(currST)$
$currST = next\_state(lastTR)$
if $\pi(lastTR) == RESET$ ***then***
$st\_reset(curST)$
end if
return *true*

The *postcondition*() retrieves the last enabled transition in $lastTR$ and ensures that the current state is a valid state in the HFSM. This is followed by immediately executing the

commit actions on the last enabled transition. This is appropriately called in the postcondition because the transition is so to say "about to be taken" such that the HFSM traverses to the next state. Therefore, the exit actions of the state must also be triggered using the $exec\_exit\_actions(currST)$. The state of the HFSM is then changed to the state pointed by the $lastTR$ transition. However, if this transition is of type $RESET$, then it is necessary to reset the destination state's refinements, which is done using the $st\_reset(currST)$. The $postcondition()$ returns with *true* value.

Definition 4.4.15. ***cleanup(...)*** *– Cleanup all entities within the HFSM.*

$\forall\ \alpha \in \{ST \cup TR\}$ ***do***
  $clean\_up(\alpha)$
return *true*

Definition 5 invokes $clean\_up()$ on all states, transitions, their refinements and so on. This is usually invoked when the HFSM completes its execution and the program is about to terminate. Thus, the allocated resources are freed in this member function.

4.5 Implementation of Hierarchical FSMs

The first requirement to fulfill is a medium of representing HFSMs. For this, we implement a graph library explained in a later chapter and create a generic template `moc_graph` that allows us to implement any directed graph-like structure. We detail our implementation of the classes for the HFSM representation for HFSMs in the following subsection.

4.5.1 Graph Representation Using Boost Graph Library

From the requirements, we see the need for implementing a comprehensive infrastructure for constructing graphs for HFSMs. In fact, many other MoCs to come may be described as graphs such as the SDF and CSP MoCs. For hierarchy, it is crucial that the graph infrastructure supports hierarchical graphs. For this purpose, we use the open-source Boost Graph Library (BGL) to develop our graph infrastructure [88].

Figure 4.5.1 shows the class diagram for implementing the graph representation generic enough such that any MoC can employ this representation. In Figure 4.5.1, `object` is an abstract base class providing access to unique identifying names for every vertex and edge. The `bgl_vertex` and `bgl_edge` classes represent vertices and edges of a graph respectively. BGL graphs can associate properties of either the vertices or edges via internal or external properties. Internal properties are used when the lifetime of the graph object is the same as the lifetime of the vertex or edge properties. On the other hand, external properties can exist even when the graph does not exist and needs additional infrastructure for maintaining the

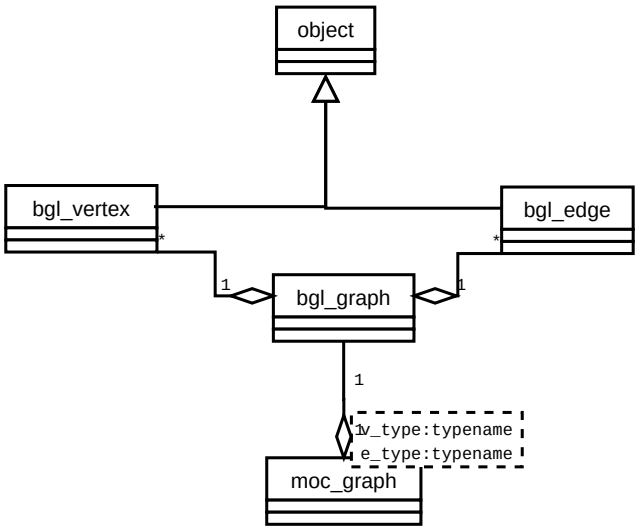


Figure 4.5: BGL Graph Package

property information such as hash tables, bidirectional hashes and so on. We employ external properties for storing properties of vertices and edges. The type of graph structure chosen is of type `boost::adjacency_list`. There are two types of graphs supported by BGL, the `adjacency_list` and the `adjacency_matrix`. The former internally uses a two-dimensional structure whereas the latter uses a matrix. The `adjacency_matrix` class is mainly used for sparse graphs. We implement the `bgl_graph` class with the functionality of inserting vertices and edges and retrieving information from the graph.

Though the implementation of the `bgl_graph` provides a class for a graph representation, it is not ready to be used by different MoCs. Therefore, we create class `moc_graph`, which is a template class with two template parameters being the type of the vertices and the edges. So, any MoC that requires using the graph library can then either inherit or contain the `moc_graph` class. This puts a restraint that every representation of a vertex or edge by an MoC must derive from `bgl_vertex` and `bgl_edge` respectively.

4.5.2 Graph Representation for HFSMs

Figure 4.6 shows the important classes reused from the graph package and the main classes in the HFSM package. The `bgl_edge` class represents the edges or arcs of a graph and the `bgl_vertex` represents the vertex or node of a graph. The `moc_graph` class is a templated class that contains the basic graph functions such as traversing through the graph, inserting vertices and edges to it and so on. The FSM MoC requires a notion of states and transitions depicted by the `fsm_state` and `fsm_transition` classes in Figure 4.6. We mandate the classes that describe the vertices and the edges of a graph to inherit the `bgl_vertex` and

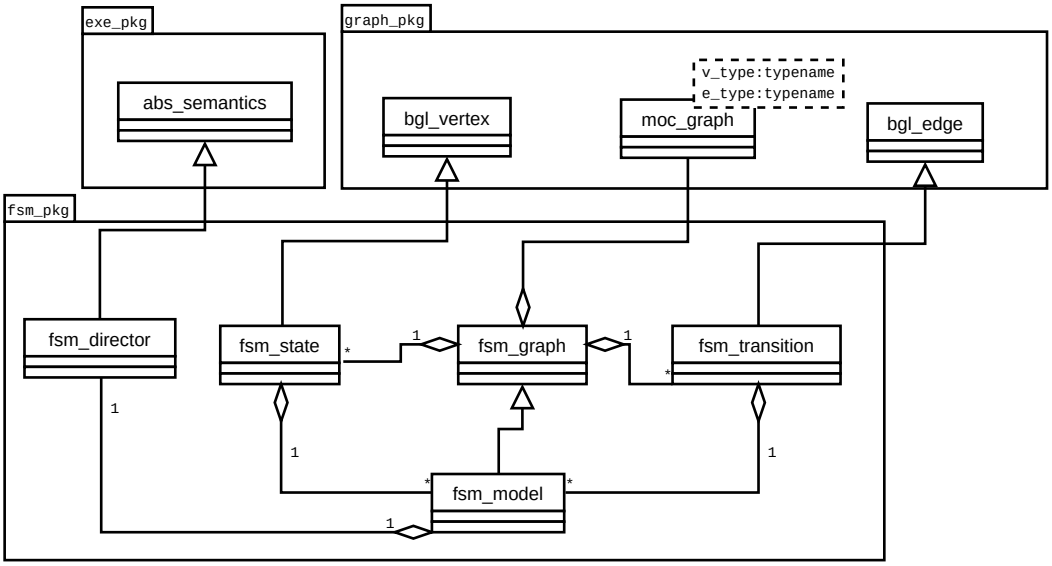


Figure 4.6: Class Diagram for HFSM Package

`bgl_edge` classes respectively, which is also shown in Figure 4.6.

The `fsm_graph` contains an object of `moc_graph` declared in the following way: `moc_graph-<fsm_state, fsm_transition>`. `fsm_graph` encapsulates all the FSM specific implementation. To allow hierarchical FSMs, either an `fsm_state` or `fsm_transition` can contain within itself one or more FSMs. To facilitate this hierarchy, we create `fsm_model` class that inherits `fsm_graph` and contains an instance of an `fsm_director`. The role of the director is to govern the execution of the model which means all execution semantics are implemented within the `fsm_director` class. One `fsm_model` can only have one `fsm_director`. However, a state or transition can have multiple FSM refinements within it. Therefore, `fsm_state` and `fsm_transition` can contain more than one `fsm_model` within it.

States & Transitions

We present snippets of the source code explaining the roles of the member functions in the classes. We will generally only present the class definitions and not discuss in depth the implementation of the classes. This will suffice in giving an idea of the data-structures and member functions used in enabling behavioral hierarchy in SystemC.

Listing 4.2 shows the class definition `fsm_state` inheriting from `bgl_vertex` and `abs_semantics` in [Listing 4.2, Line 1]. Figure 4.6 also shows this inheritance relationship. Inheriting the `abs_semantics` class means that `fsm_state` class must provide implementations for the pure virtual members in `abs_semantics`. We do not show them listed in this snippet but and we mark them being defined at [Listing 4.2, Line 20]. The start of the public definition

Listing 4.2: `fsm_state` Class Definition

```
1 class fsm_state : public bgl_vertex,  public abs_semantics
2 {
3 public:
4   /// typedefss for user hierarchy functions for list
5   typedef abs_list< fsm_model > refinement_t;
6   typedef vector< fsm_model *> hierarchy_fsms_t;
7   /// User can override these functions
8   virtual bool entry_actions();
9   virtual bool exit_actions();
10  /// Run-to-complete member functions
11  bool is_run_complete();
12  void set_run_complete();
13  /// Reset the refinements
14  bool reset_hierarchy();
15  /// FSM hierarchy
16  void add_fsm_refinement( fsm_model * _fsm );
17  bool has_hierarchy();
18  hierarchy_fsms_t  get_fsm_refinement_models();
19  /// Overload members of abstract semantics
20  /// ... more code
21  /// Default, overloaded constructors
22  /// Overloaded operators
23  /// ... more code
24 private:
25  /// List of hierarchical FSMs
26  refinement_t fsm_refs;
27  bool run_complete;
28  refinement_t * get_fsm_refinement();
29  void elaboration( fsm_model * _fsm );
30 };
```

begins with two `typedef`s. We use this coding style to avoid repeating complex types, which in this case are `abs_list<fsm_model>` and the `vector<fsm_model>`. The `abs_list` class is a utility class explained in the chapter that explains the graph package. However, its basic purpose is to provide a medium through which the abstract semantics functions can be invoked on all the refinements. The `hierarchy_fsms_t` type is used to hold pointers to the refinements within a state. The `refinement_t` type is declared at [Listing 4.2, Line 26] for the data member `fsm_refs` that holds the HFSM refinements in this state. Note that the class `fsm_model` is not defined as yet and there is simply a forward class declaration used to allow compilation of the `fsm_state` class. The `entry_actions()` and `exit_actions()` shown in [Listing 4.2, Line 8 & 9] can be overridden by the user if the user desires to employ the capability of entry and exit actions. The `set_run_complete()` and `is_run_complete()` member functions set the `run_complete` private data member in [Listing 4.2, Line 27] and returns the Boolean value, respectively. When the `run_complete` Boolean data member is set, then the refinements in this state must follow the run-to-complete semantics.

The resetting of refinements within a state is done using `reset_hierarchy()` in [Listing 4.2, Line 14]. This member function is usually invoked when a transition is set to the `RESET`

type, which requires resetting the destination state's refinements. Adding refinements into a state is done through the `add_fsm_refinement()` member function. This inserts a pointer to the HFSM into the `fsm_refs` data member. However, before doing that, we ensure certain requirements on the added refinement, such as that the `_fsm` pointer cannot be `NULL` and we check these using the private member function `elaboration()` [Listing 4.2, Line 29]. We define a member function that returns a Boolean value of `true` indicating that the state has at least one refinement embedded within it via the `has_hierarchy()`. To retrieve refinements, we have the `get_fsm_refinement_models()` member function in [Listing 4.2, Line 18], which returns a list of all the HFSMs in that state. We use an additional private member function `get_fsm_refinement()` in [Listing 4.2, Line 28] to get a pointer of type `refinement_t` such that members from the abstract semantics can be invoked on all the refinements in the state.

Listing 4.3: `fsm_transition` Class Definition

```
1 class fsm_transition : public bgl_edge, public abs_semantics {
2 public:
3   typedef abs_list< fsm_model > refinement_t;
4   typedef vector< fsm_model *> hierarchy_fsms_t;
5   virtual bool guard() ;
6   virtual bool choice_actions();
7   virtual bool commit_actions();
8   bool is_enabled();
9   bool is_preemptive();
10  bool is_reset();
11  bool is_run_complete();
12  void set_enabled();
13  void set_preemptive();
14  void set_reset();
15  void set_run_complete();
16  bool reset_hierarchy();
17  void add_fsm_refinement( fsm_model * _fsm );
18  bool has_hierarchy();
19  hierarchy_fsms_t  get_fsm_refinement_models();
20  /// Overload members of abstract semantics
21  /// ... more code
22  /// Default, overloaded constructors
23  /// Overloaded operators
24  /// ... more code
25 private:
26  refinement_t fsm_refs;
27  bool enabled;
28  bool preemptive;
29  bool reset;
30  bool run_complete;
31  void elaboration( fsm_model * _fsm );
32  refinement_t * get_fsm_refinement();
33 };
```

The `fsm_transition` class represents the transitions that connect the states of type `fsm_state`. As depicted in Figure 4.6, the `fsm_transition` class inherits from the `bgl_edge` and `abs_s-`

`emantics` classes. The `bgl_edge` class is a generic base class for representing edges in a graph and the `abs_semantics` the abstract semantics associated with an executable entity. The virtual member functions in [Listing 4.3, Line 5 - 7] show the `guard()`, `choice_actions()` and the `commit_actions()`. The `guard()` member function requires the designer to test the guard associated with this transition and return a `true` in the case where the guard is satisfied and `false` when it is not. The return Boolean value is used in the kernel to trigger the actions associated with this transition. Depending on whether the guard is satisfied, the `choice_actions()` or the `commit_actions()` are executed. Member functions to get the properties of the transition are shown in [Listing 4.3, Line 8 - 11] and those that set the properties in [Listing 4.3, Line 12 - 15]. These get and set member functions check the type of the transition such as `PREEMPTIVE`, `RESET` or `RUN-TO-COMPLETE`. Their respective private data members are shown in [Listing 4.3, Line 27 - 30]. The transitions can also embed other HFSM refinements and thus we have the `add_fsm_refinement()` member function that allows inserting refinements in a transition.

4.5.3 HFSM Graph Representing the HFSM Model

The generic graph package implements an `moc_graph` templated class, which allows any MoC to easily implement their graph-like structure. The `fsm_graph` class aggregates the `moc_graph` class and renders the class with member functions specific for describing an FSM. Listing 4.4 displays the source code listing for the `fsm_graph` class definition.

The `fsm_graph` `typedef`s a list of `fsm_state` and `fsm_transition` objects with `state_list` and `transition_list`. These are used as return types for retrieving all the states and transitions in the FSM using the `get_all_states()` and `get_all_transitions()` member function in [Listing 4.4, Line 7 & 8]. The `set_init_state()` member function sets the initial state for the HFSM, which assigns the pointer shown in [Listing 4.4, Line 28]. Similarly, the `set_final_state()` and the `set_curr_state()` set the `final_st` and `curr_st` private members in [Listing 4.4, Line 29 & 30]. Given a transition, the `get_target()` member function returns the destination state of that transition. However, to retrieve the states that are connected to a particular state via some transition, we employ the `get_adjacent_states()`. The return type of this is once again a list of `fsm_state` pointers. To retrieve all the outgoing transitions from a state, we use the `get_outgoing_transitions()` member function, which returns a `transition_list`. The `get_num_states()` and `get_num_transitions()` return the number of states and transitions in the HFSM. Until now, the member functions described are used for manipulating the information present in the HFSM. However, to insert states and transitions into the HFSM we provide the `insert_state()` and `insert_transition()` member functions. We have overloaded these members so as to allow passing by reference as well as by pointer. The insertion of transition requires three arguments and they are the `fsm_state` from which the transition is outgoing and the `fsm_state` to which the transition is the destination and lastly the transition to be associated with the `from` and `to` `fsm_state` objects as shown in [Listing 4.4, Line 21]. Finally, the private member shown in [Listing

Listing 4.4: `fsm_graph` Class Definition

```
1 class fsm_graph
2 {
3 public:
4   typedef vector< fsm_state* > state_list;
5   typedef vector< fsm_transition* > transition_list;
6   /// Functions only specific for fsm_graph
7   state_list get_all_states() ;
8   transition_list get_all_transitions() ;
9   void set_init_state( fsm_state * _st);
10  void set_final_state( fsm_state * _st);
11  void set_curr_state( fsm_state * _st );
12  fsm_state * get_init_state();
13  fsm_state * get_final_state();
14  fsm_state * get_curr_state();
15  fsm_state * get_target( const fsm_transition & _tr );
16  state_list get_adjacent_states( const fsm_state & _st );
17  transition_list get_outgoing_transitions( const fsm_state & _st);
18  unsigned int get_num_states() ;
19  unsigned int get_num_transitions() ;
20  void insert_state(fsm_state & _st);
21  void insert_transition( const fsm_state & from, const fsm_state & to,
        fsm_transition & _tr);
22  void insert_state(fsm_state * _st);
23  void insert_transition( const fsm_state * from, const fsm_state * to,
        fsm_transition * _tr);
24  fsm_graph();
25  ~fsm_graph();
26 private:
27  moc_graph< fsm_state, fsm_transition > fsm_g;
28  fsm_state *init_st;
29  fsm_state *final_st;
30  fsm_state *curr_st;
31 };
```

4.4, Line 27] is an aggregation of the `moc_graph` class. The `fsm_g` private data member is of type `moc_graph` with the vertices being `fsm_state` type and the edges being `fsm_transition` type.

Capturing the graph-like structure of the HFSM using `fsm_graph` only allows a way to represent the HFSM, but it does not implement the abstract semantics or any real semantics associated with it. Since we wish to keep the semantic representation separate from the model representation, we implement the semantics or the HFSM kernel in the `fsm_director` class. However, keeping heterogeneous behavioral hierarchy in mind, we need to associate one instance of the simulation kernel to one level of hierarchy of a model. To do this, we implement another class called `fsm_model` that represents one HFSM model that has an association with the `fsm_director` class as shown in [Listing 4.5, Line 18]. This class mainly implements accessory member functions alongside the abstract semantics. The `check_transitions()` member function verifies that only one transition is enabled because otherwise the HFSM would express non-determinism, which we disallow in

Listing 4.5: `fsm_model` Class Definition

```
1 class fsm_model :  public fsm_graph, public abs_semantics
2 {
3 public:
4   /// fsm_model specific functions
5   int check_transitions( transition_list & tl );
6   bool process_transitions();
7   fsm_transition * get_enabled_transition();
8   bool stop();
9   bool trigger();
10  bool iterate(); /// precondition, execute, postexecute
11  void reset_last_enabled_transition();
12  /// Overload members of abstract semantics
13  /// ... more code
14  /// Default, overloaded constructors
15  /// Overloaded operators
16  /// ... more code
17 private:
18  fsm_director fsm_dir;
19  fsm_transition * enabled_tr;
20 }; /// fsm_model class
```

our realization of the HFSM MoC. The `process_transitions()` member function uses the `check_transitions()` to ensure determinism. We incorporate a `stop()` member function that can be used to halt the execution of the FSM. The remainder of the member functions such as `trigger()` and `iterate()` were explained in the formalization of the execution semantics. The `reset_last_enabled_transition()` simply resets the last enabled transition to its default.

Listing 4.6: `fsm_director` Class Definition

```
1 class fsm_director :  public abs_semantics
2 {
3 public:
4   void set_fsm_graph(fsm_model * _fsm);
5   bool iterate();
6   bool iterate_refinements_state_rtc( fsm_state * _st );
7   bool iterate_refinements_transition_rtc( fsm_transition * _tr );
8   /// Overload members of abstract semantics
9   /// ... more code
10  /// Default, overloaded constructors
11  /// Overloaded operators
12  /// ... more code
13 private:
14  fsm_model * model;
15  bool reexecute;
16 }; /// class fsm_director
```

The actual kernel implementation is done in the `fsm_director` class. One instance of the kernel only handles one level of the HFSM. Therefore, the kernel is associated with only one

instance of the `fsm_model` object as shown in [Listing 4.6, Line 14]. This pointer is assigned its `fsm_model` during the initialization of the HFSM, which includes instantiating an object of `fsm_director` and then invoking the `set_fsm_graph()` member function in [Listing 4.6, Line 4] to pass the address of the `fsm_model` object. The `iterate()` member function again only invokes the `precondition()`, `execute()` and `postcondition()` of the abstract semantics. We also show specific iterate member functions for the state and transition refinements with the run-to-complete property in [Listing 4.6, Line 6 & 7]. These two member functions exhibit the same functionality except one is for refinements in states and the other in transitions. The member functions perform iterations on all the refinements until the HFSM reaches its final state. Once the HFSM reaches the final state, the `reexecute` Boolean private data member is set such as to inform the kernel that another iteration is not required for the `RUN-TO-COMPLETE` property refinements. The remainder of the abstract semantics actually implement the algorithms presented in the definitions earlier. We do not present the implementation details regarding the actual algorithm as we have already done so formally.

4.6 Modeling Guidelines for HFSM

This section provides code snippets showing the programming style that can be used to model HFSM using the extension. The code snippets are for the example of the sampler presented in Figure 4. This sampler component shown in Figure 4.6 is a part of the power model which will be explained in section 4.7.

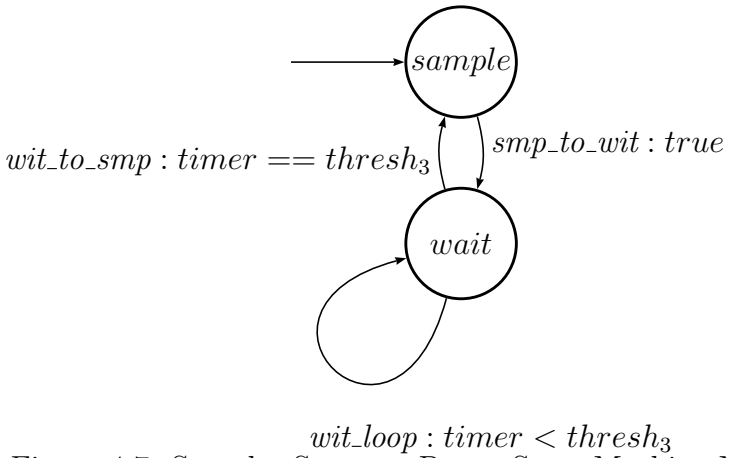


Figure 4.7: Sampler Segment Power State Machine Model

Listing 4.7 shows implementation of the transitions used in the sampler component. Note that for brevity we provide functionality of all the transitions into one transition class

Listing 4.7: Sampler's Transition Definition

```
1 SCH_FSM_TRANSITION( sampler_tr )
2 {
3  public:
4   SCH_FSM_TRANSITION_CTOR( sampler_tr )
5     {
6       thres_h = 3; id = _str;
7     };
8
9   SCH_FSM_TRANSITION_GUARD()
10    {
11      bool flag = false;
12      if ( id == "smp_to_wit" )
13        flag = true;
14      else
15        if ( id == "wit_loop" )
16          {
17            if ( counter.read () < thres_h )
18              flag = true;
19            else
20              flag = false;
21          }
22        else
23          if ( id == "wit_to_smp" )
24            {
25              if ( counter.read() == thres_h )
26                flag = true;
27              else
28                flag = false;
29            }
30          else
31            flag = false;
32      return flag;
33    };
34
35  SCH_FSM_COMMIT_ACTIONS()
36    {
37      return true;
38    };
39
40   behavior_interface < bool > task_arr;
41   behavior_interface< sc_int<8> > counter;
42   string id;
43   sc_int<8> thres_h;
44 };
```

`sampler_tr`. We begin by defining a class `sampler_tr` using the MACRO shown in [Listing 4.7, Line 1]. The expansion of this MACRO is simple in that it inherits from the `fsm_transition` class. This is followed by the `sampler_tr` class's constructor. We require that every transition's constructor pass a unique string identifier for identifying that particular object. The MACRO expands with an argument variable `_str`, which we use to set the `id` data member in [Listing 4.7, Line 6]. Since we are trying to compact all transitions into this one class, we use the `id` to identify the unique transitions, which can be seen in the guard definition in [Listing 4.7, Line 9 - 33]. We append Figure 4.6 with transition labels used in the implementation. For example, ***wit_to_smp*** : $timer == thresh_3$ shows the transition going from the `wait` state to the `sample` state with the label ***wit_to_smp*** and the guard being $timer == thresh_3$. There is no guard on the transition going from the `sample` to `wait` state shown by ***smp_to_wit***. The corresponding source code is shown in [Listing 4.7, Line 12 & 13]. We check for the unique string identifier and simply set the `flag` to `true`, which is returned at the end of the guard definition in [Listing 4.7, Line 32]. The self-loop on the `wait` state labeled ***wit_loop*** is only enabled after a certain threshold `thres_h` ([Listing 4.7, Line 15 - 21]). The ***wit_to_smp*** transition similarly is enabled only when the threshold has been reached. In this example we do not perform any specific actions and simply return `true` for the commit actions in [Listing 4.7, Line 35]. Notice the `behavior_interface` definitions in [Listing 4.7, Line 40 & 41]. We provide two utility classes called `behavior_interface`, which are synonymous to ports and `behavior_tunnel` that are synonymous to channels. We must ensure that when using the SystemC extensions, the DE semantics is not employed thus disallowing the use of `sc_signal` channels or `sc_port` declarations.

We require the designer to create a model for the HFSM and Listing 4.8 shows an example of the sampler model. We use the `SCH_FSM_MODEL()` MACRO to define a `sampler` model class. We begin by defining pointers to the states and transitions that we instantiate in the constructor of the `sampler` model as in [Listing 4.8, Line 4 - 8]. The data member `timer_mod` in [Listing 4.8, Line 10] is the timer component's definition. We use this as a refinement in the `wait` state. So, the model constructor that uses the `SCH_FSM_MODEL_CTOR()` MACRO, contains the instantiations for the data members. We employ the `insert_state()` member function to insert the states and the `insert_transition()` to connect and insert transition into the HFSM model. Examples of these invocations are in [Listing 4.8, Line 21 & 23]. the `insert_transition()` member function takes in three arguments. The first argument is the state from which the transition goes out from, the second is the state to which the transition points towards and the third is the transition used to connect the two states. We use the `set_init_state()` member function to set the initial state to the `sampler` state in [Listing 4.8, Line 33]. The `add_fsm_refinement()` embeds an FSM refinement in either a state or a transition and in [Listing 4.8, Line 34] adds the refinement in the `wait` state. In addition, we also have a transition with the `RESET` type shown in [Listing 4.8, Line 35]. Further examples are available at our website [86].

Listing 4.8: Sampler's Model Definition

```
1 SCH_FSM_MODEL( sampler )
2 {
3  public:
4   fsm_state * smp;
5   fsm_state * wit;
6   sampler_tr * smp_to_wit;
7   sampler_tr * wit_loop;
8   sampler_tr * wit_to_smp;
9
10  timer_model * timer_mod;
11
12  SCH_FSM_MODEL_CTOR( sampler )
13    {
14      smp = new fsm_state("sample" );
15      wit = new fsm_state("wait");
16
17      smp_to_wit = new sampler_tr( "smp_to_wit");
18      wit_loop = new sampler_tr("wit_loop");
19      wit_to_smp = new sampler_tr("wit_to_smp");
20
21      insert_state( *smp );
22      insert_state( *wit );
23      insert_transition( *smp, *wit, *smp_to_wit );
24      insert_transition( *wit, *wit, *wit_loop );
25      insert_transition( *wit, *smp, *wit_to_smp);
26
27      timer_mod = new timer_model();
28      wit_loop->counter( *timer_mod->counter );
29      wit_loop->thres_h = 3;
30      wit_to_smp->counter( *timer_mod->counter );
31      wit_to_smp->thres_h = 3;
32
33      set_init_state( smp );
34      wit->add_fsm_refinement ( timer_mod );
35      smp_to_wit->set_reset();
36    };
37  ~sampler();
38 };
```

4.7 HFSM Example: Power Model

We present the full example of the power model in Figure 5.7. Additional examples are available at our website [86]. 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 change of state. 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 4.7.

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

Table 4.1: Simulation Speeds for Power Model

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 gain increased modeling fidelity and we are exercising behavioral hierarchy. However, this is promising for heterogeneous behavioral hierarchy which we see in the following chapters.

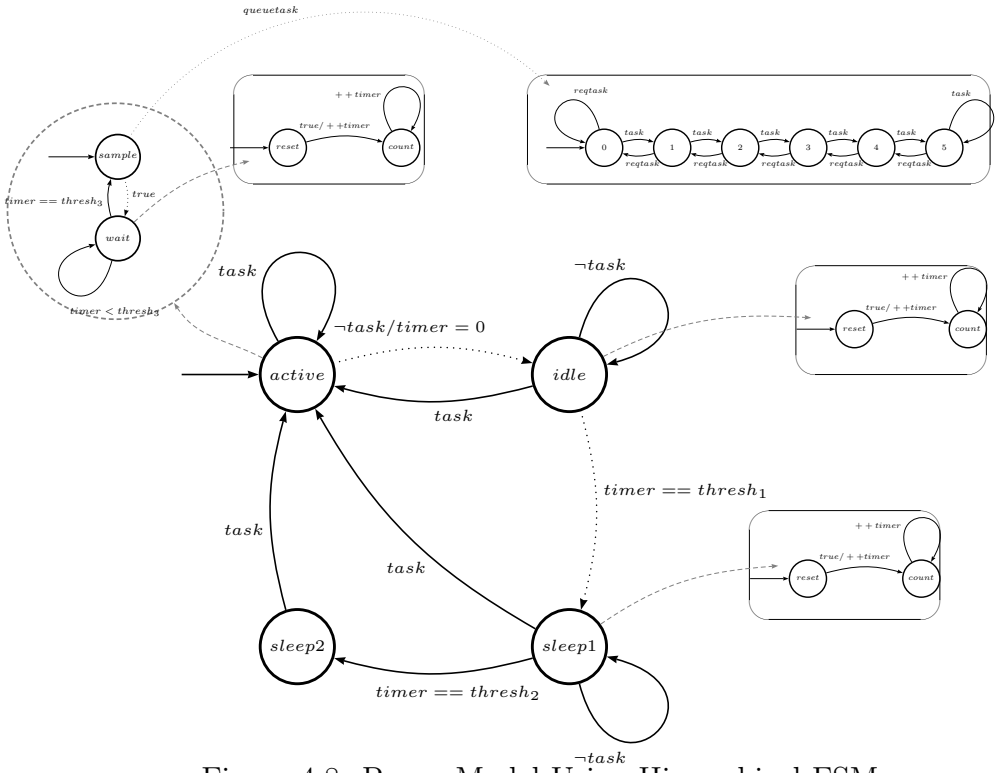


Figure 4.8: Power Model Using Hierarchical FSMs

Chapter 5

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 invokes 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 Figure 5.2. This formal representation is evidence for 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 Figure 5.1. We borrow the SDF semantics from SystemC-H and point the readers to [26, 33, 32] 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.

5.1 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 5.1. 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 example, the `prepare()` of the `sdf_model` is responsible for computing the schedule as shown in Figure 5.1, which only executes once, after which iterations of the entity are performed. Note in Listing 5.1 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 5.1: 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 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.

5.2 Basic Definitions

The basic definitions constitue of defining the sets for FSM states and transitions and SDF block objects, functions to check if an object has a refinement, to distinguish the type of the object, whether the run-to-complete property of an object is set and to iterate through all

refinements of an object. Please note that we redefine these basic definitions when discussing our implementation.

Definition 5.2.1. *Let β be the set of SDF block objects*
Let $S \subset \beta$ be the set 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 5.2.2. *To distinguish the type of an element.*

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

Definition 5.2.3. *Check if entity has refinement.*

$$\rho(x) = \begin{cases} true & x \text{ has at least one refinement} \\ false & x \text{ has no refinements} \end{cases}$$
$$where\ x \in \{\beta \cup ST \cup TR\}$$

Definition 5.2.4. *Check if object is set to run-to-complete*

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

Definition 5.2.5. $\boldsymbol{\gamma}(...)$ – *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 $x \in \{\beta \cup ST \cup TR\}$
Let $iterate(x)$ perform an iteration on the refinement of object x

$$\boldsymbol{\gamma}(x) = \begin{array}{l} \textbf{\textit{Require:}}\ R = \emptyset \\ R = get\_all\_refs(x) \\ \forall\ r \in R\ \textbf{\textit{do}} \\ \quad \underline{iterate(r)} \end{array}$$

Algorithm 3 Algorithm for Execution Semantics for HFSMs

```
α = get_enabled_tr(currST)
if α ≠ ∅ then
  if ρ(α) = true then
    γ(α)
  end if
  execute_choice_action(α)
end if
{Execute state refinements}
if ρ(currST) = true then
  γ(curST)
end if
{Execute this before state change}
if α = get_last_enabled_tr() then
  execute_commit_action(α)
  currST = next_state(α)
  if π(α) = RESET then
    reset_state(currST)
  end if
end if
```

5.3 Execution Semantics for Starcharts

We describe the *charts execution semantics using the algorithm presented in algorithm 3. However, to provide an intuitive understanding of hierarchical FSMs, let us take the simple example shown in Figure 4.2 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 refinements 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 3, 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 *charts. 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 $\gamma()$ function. Once the iterations are completed, the choice action of the enabled transition is triggered.

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 Figures 5.1 and 5.2 using the abstract symantics we presented in Figure 5.1.

Definition 5.3.1. ***sdf_director's prepare(...)*** *– Prepare SDF block objects in director.*

Require: $S = \emptyset$, $\beta \neq \emptyset$
  $\forall\ \alpha \in \beta$ ***do***
    $\underline{prepare(\alpha)}$

  $S = \Gamma(\beta)$
  return *true*

Definition 5.3.2. ***sdf_director's precondition(...)*** *– Precondition for SDF director.*

  return *true*

Definition 5.3.3. ***sdf_director's execute(...)*** *– Execution of an SDF director.*

Require: $S \neq \emptyset$
  $\forall\ \alpha \in S$ ***do***
    if $\rho(\alpha) = true$ ***then***
      $\underline{\gamma(\alpha)}$
    end if
    $\underline{execute(\alpha)}$

  return *true*

Definition 5.3.4. ***sdf_director's postcondition(...)*** *– Postcondition for SDF director.*

  return *true*

Definition 5.3.5. ***sdf_director's cleanup(...)*** *– Cleanup for SDF director.*

  clean_up()
  return *true*

Figure 5.1: Simulation Algorithm: Starting from a Heterogeneous Hierarchical SDF Model

Definition 1. ***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***
    $\underline{prepare(\alpha)}$

  return *true*

Definition 2. ***fsm_director's precondition(...)*** – *Precondition for FSM model.*

  return *true*

Definition 3. ***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***
    $\underline{\gamma(\alpha)}$
    $execute\_choice\_actions(\alpha)$
    return *true*
  end if
  $\underline{\gamma(currST)}$
  if $\pi(\alpha) = NONPREEMPTIVE$ *&* $\rho(\alpha) = true$ ***then***
    $\underline{\gamma(\alpha)}$
    $execute\_choice\_actions(\alpha)$
  end if
  return *true*

Definition 4. ***fsm_director's postcondition(...)*** – *Postcondition for FSM director.*
Let lastTR be the last enabled transition
where $lastTR \in \{TR \cup \emptyset\}$
Require: $lastTR$, $currST \neq \emptyset$
  $execute\_commit\_actions(lastTR)$
  $currST = next\_state(lastTR)$
  if $\pi(lastTR) = RESET$ ***then***
    $st\_reset(curST)$
  end if
  return *true*

Definition 5. ***fsm_director's cleanup(...)*** – *Cleanup for FSM director.*
  $clean\_up()$
  return *true*

Figure 5.2: Simulation Algorithm: Starting from a Heterogeneous Hierarchical FSM Model

5.4 Our Execution Semantics for Hierarchical FSMs

Continuing to use the definitions presented earlier, we provide few additional definitions that allows us to present the our algorithm for the execution semantics of HFSMs. Our algorithm is a variant of the *charts 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.

5.4.1 Additional Definitions Specific for HFSMs

The $rtc\_iterate()$ function performs run-to-complete iterations on the object x. In the current definition, it invokes a function that is specific to the HFSM MoC, $rtc\_fsm\_iterate()$. We define $get\_all\_ref()$ that returns all the refinements in an object and $\pi()$ to determine the different types of possible transitions. Lastly, we redfine the $\gamma()$ function to incorporate run-to-complete iterations into the execution semantics.

Definition 5.4.1. *Run-to-complete iterate for different HFSMs*

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

Definition 5.4.2. *Return refinements for particular object of different MoC.*

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

Definition 5.4.3. *To destinguish the type of transition object.*

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

$$where\ x \in TR$$

Definition 5.4.4. $\boldsymbol{\gamma}(...)$ – *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 $x \in \{\beta \cup ST \cup TR\}$
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

$\boldsymbol{\gamma}(x) =$

Require: $R = \emptyset$
$R = get\_all\_refs(x)$
$\forall\ r \in R$ ***do***
if *run_to_complete(r)*
then
rtc_iterate(r)
else
iterate(r)
end if

5.4.2 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 5.4.5. *Compute the executable schedule for an SDF graph.*

$$\Gamma(X) = X'\ if\ X' \in \beta\ where\ X \subseteq \beta$$

Definition 5.4.6. *Run-to-complete iterate for different MoCs*

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

Definition 5.4.7. *Return refinements for particular object of different MoC.*

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

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 $get\_all\_ref()$ 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 Figure 5.1 and Figure 5.2 unaltered. In particular note the underlined invocations because these are responsible for giving control to a refinement's respective simulation kernel. For example, in Figure 5.1 the call to $\gamma()$ 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.

5.5 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. Firstly, 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 graph-like 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.

5.5.1 Graph Representation Using Boost Graph Library

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) [88] and reuse this graph library for the FSM and SDF MoCs. For enabling hierarchy, it is crucial that the graph infrastructure supports hierarchical graphs. Figure 4.5.1 shows the class diagram for the generic graph library.

Figure 4.5.1 shows the class diagram for implementing the graph representation generic enough such that any MoC can employ this representation. The `object` class is an abstract base class providing access to unique identifying names for every vertex and edge. The `bgl_vertex` and `bgl_edge` classes represent vertices and edges of a graph respectively. BGL graphs can associate properties of either the vertices or edges via internal or external properties. Internal properties are used when the lifetime of the graph object is the same as the lifetime of the vertex or edge properties. On the other hand, external properties can exist even when the graph does not exist and needs additional infrastructure for maintaining the property information such as hash tables, bidirectional hashes and so on. We employ external properties for storing properties of vertices and edges. The type of graph structure chosen is of type `boost::adjacency_list`. There are two types of graphs supported by BGL, the `adjacency_list` and the `adjacency_matrix`. The former internally uses a two-dimensional structure whereas the latter uses a matrix. The `adjacency_matrix` class is mainly used for sparse graphs. We implement the `bgl_graph` class with the functionality of inserting vertices and edges and retrieving information from the graph.

Though the implementation of the `bgl_graph` provides a class for a graph representation, it is not ready to be used by different MoCs. Therefore, we create class `moc_graph`, which is a template class with two template parameters being the type of the vertices and the edges. So, any MoC that requires using the graph library can then either inherit or contain the `moc_graph` class. This puts a restraint that every representation of a vertex or edge by an MoC must derive from `bgl_vertex` and `bgl_edge` respectively.

Enabling SystemC with Hierarchical FSMs

Figure 4.6 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.

5.5.2 Implementing the Execution Semantics

Figure 5.2 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 Figure 5.2 follows the semantics defined earlier in section 5.3 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. When enabled, the preemptive transition skips execution of the current state's refinement. The reset transition during the `postcondition()` resets the refinement of the destination state. The nonpreemptive transition executes the refinement of the transition, followed by executing the current state's refinement and then enters `postcondition`. The run-to-complete semantics allows the particular refinement to execute until the FSM reaches the final state.

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 commit actions is when 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 Figure 5.2, `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.

Enabling SystemC with Hierarchical SDFs

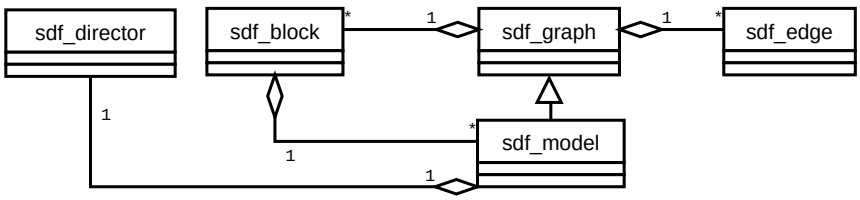


Figure 5.3: SDF Library

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. Figure 5.5.2 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 FIFOs 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.

Enabling SystemC with Hierarchical SDFs & FSMs

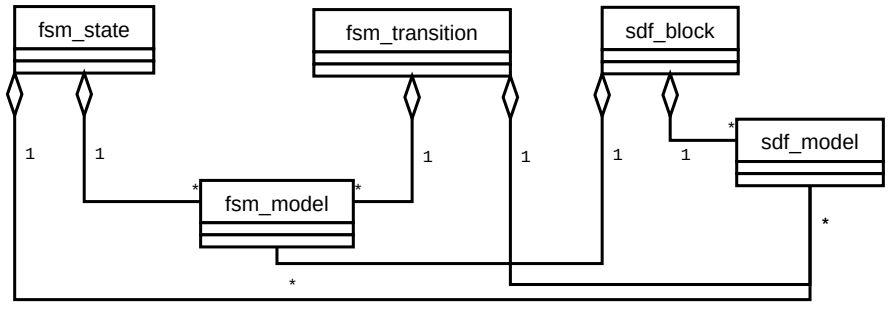
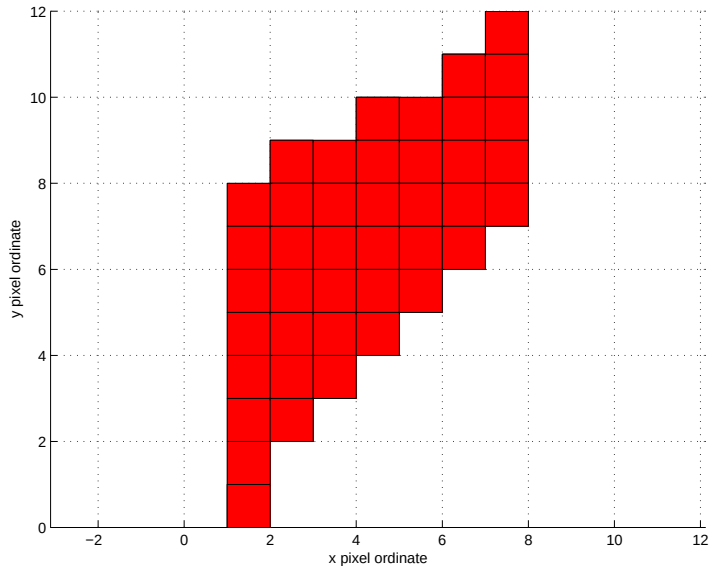


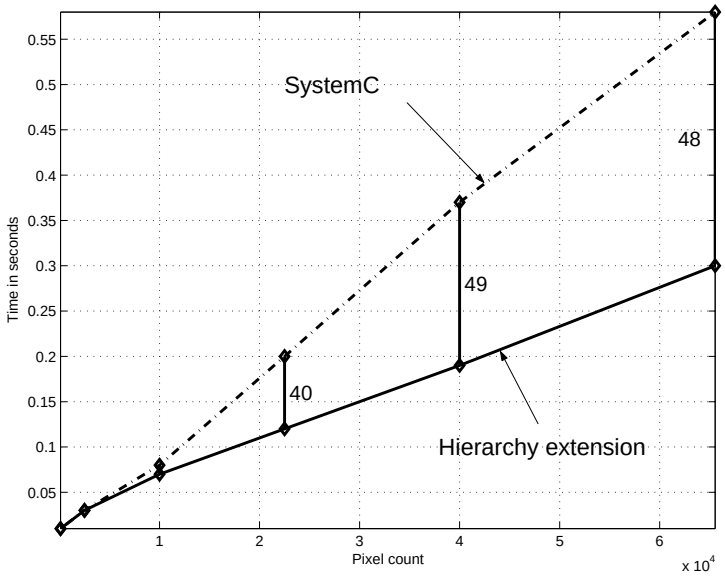
Figure 5.4: SDF and HFSM Library

Enabling heterogeneous hierarchy between SDF and FSM models is shown using snippets of the class diagram in Figure 5.5.2. 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.

5.6 Examples



(a) In-fill Processor Output



(b) Simulation Results for In-fill Processor

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

(c) Table Showing Simulation Times

Figure 5.5: Polygon In-fill Processor Results

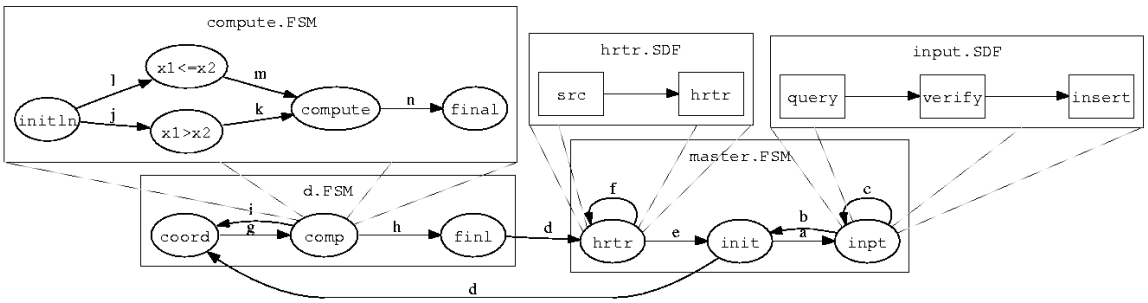


Figure 5.6: Behavioral Decomposition of In-fill Processor

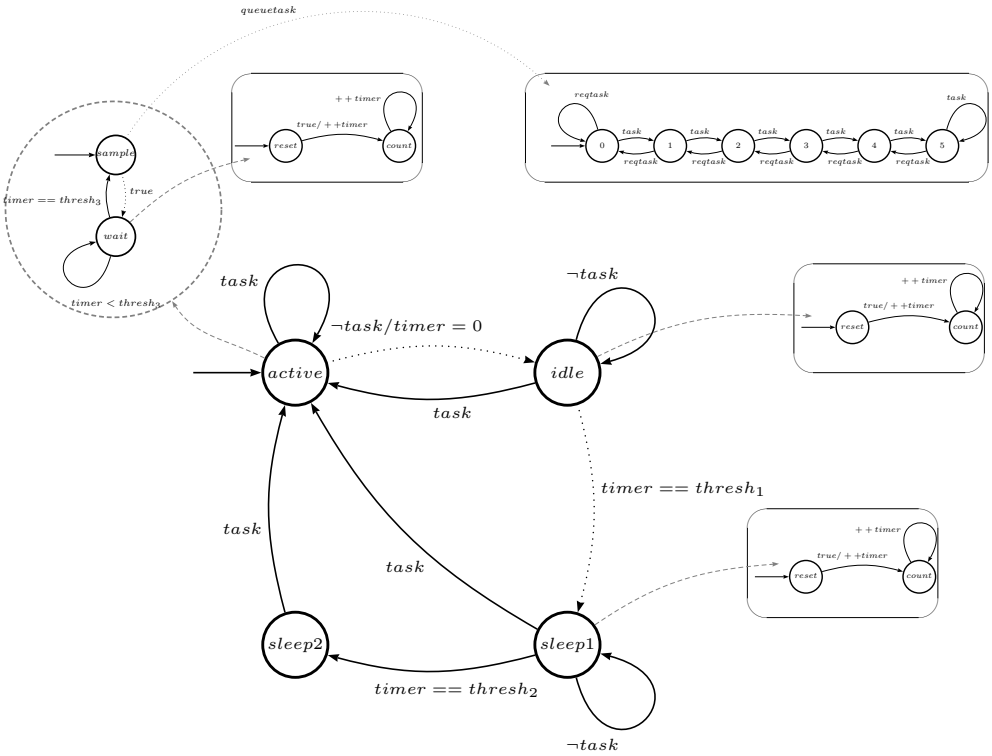


Figure 5.7: Power Model Using Hierarchical FSMs

5.6.1 Polygon In-fill Processor

This example shown in Figure 5.6 implements a variation of the polygon in-fill processor (PIP) from [16], 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 5.5(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 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 Figure 5.6 describes our implementation at a higher level of abstraction without specifics on the transition guards, production and consumption rates, or the `behavior_interface`s and `behavior_tunnel`s 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`.

User Input FSM The user specifies four vertices that define the closed region to be filled in. This component shown on Figure 5.6 as `input.SDF` is modeled as an SDF with three SDF blocks. The first block, `query` is a prompt to request pixel coordinates, the second verifies that the pixel coordinate input is within the constraints of the 256×256 pixel resolution. The third block, `insert` adds the vertex into a storage (RAM) in hardware, modeled as a global list in this model. The `input.SDF` is embedded in the master controller with transition `e` being enabled four times, causing the SDF refinement to trigger and request four coordinates defining the closed region.

Bresenham's Line Algorithm Transition `d`'s refinement selects two of the coordinates entered by the user and transitions to the `comp` state, which has a refinement. This refinement `compute.FSM` computes the Bresenham's line algorithm [16, 89] for the two coordinates. Transition `g` is a `reset` transition causing the refinement of the destination to reset on every iteration. Once the four connecting lines are computed, `compute.FSM` reaches `final` and thus causes `d.FSM` to reach `finl` completing transition `d.FSM` to state `hrtr`.

Horizontal trace The `hrtr.SDF` is an SDF model with two components. The first component selects two coordinates with the same y-ordinate representing the vertical positioned pixel and the minimum and maximum x-ordinates from the Bresenham's line algorithm. Upon correct discovery of the x-ordinates, the `hrtr` block executes traversing through all the vertical pixels and identifying the coordinates to be shaded.

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 5.5(b) 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 5.5(c) 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 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 as shown in Figure 5.5(b).

Chapter 6

Bluespec ESL and its Co-simulation with SystemC DE

Increasing difficulty in managing highly complex and heterogeneous designs of today for design space exploration has been leading to raising the level of abstraction during modeling and simulation. Some popular examples of the raised abstraction level are the introduction of SystemC [8] as a language targeted for register-transfer level (RTL) and above RTL and heterogeneity in system level design languages and frameworks (SLDL)s that support models of computation (MoC)s for representing designs [26, 5]. SystemVerilog [12] addresses adding object-oriented capabilities to Verilog's modeling and simulation framework as well. Even though these methodologies aim at providing better design productivity and simulation performance, they result in a disconnect from the model representation and its realization of the implementation model. In fact, problems such as behavioral synthesis from higher than RTL models is a difficult problem being tackled by industry tools such as Forte's Cynthesizer [90] and Celoxica's Agility compiler [91] for SystemC, Bluespec's BSC [6] for SystemVerilog, and Tensilica's XPRES [92], Mentor's Catapult [93] for C/C++ to mention a few of them. However, arguably none of these achieve full synthesizability unless a refinement into a synthesizable subset is manually undertaken.

To avoid rewriting models expressed at higher abstraction levels into a synthesizable subset, several refinement methodologies are proposed such as in Metropolis [9], ForSyDE [10], SML-Sys [11] by which an implementation model may be realized. This sometimes overburdens the designer in not only constructing an abstract model, but by also requiring the designer to later refine the model for implementation. This in-turn requires repeated validation for correct functionality, equivalence checks for ensuring abstract and implementation models are equivalent and so on.

It is known that easing the designer's experience during modeling and then later for debugging are essential for designer productivity, as is the simulation efficiency of an SLDL. Furthermore, it is important for an SLDL to provide a software development environment

with the increasing hardware/software dependencies in today's designs. However, it is of utmost importance that such an SLDL cleanly fits into a design flow going from modeling and simulation to one that leads to implementation. We feel that many SLDLs lag in this one crucial aspect of the design flow. One attractive methodology that has successfully shown to fit into the design flow is Bluespec's rule-based MoC [39, 40, 41] embedded in Verilog. The rule-based MoC has a simple yet powerful semantic model that has a direct mapping to its implementation. Evidence of this methodology's success is discussed in [39, 40, 41]. Hence, we envision extending on the success experienced by the rule-based MoC in Bluespec to one of the leading SLDLs namely, SystemC. We present a kernel-level extension to SystemC for Bluespec's rule-based MoC that is interoperable with SystemC's reference implementation and allows for a simple yet powerful co-design environment.

An argument put forward against SystemC is that current SystemC is not making the task of modeling designs any different from regular hardware description languages (HDL)s, with the exception of transaction-level modeling (TLM). Thus, using SystemC solely for RTL defeats the purpose of employing a language for modeling and simulation at higher levels. However, even at TLM, SystemC's reference implementation kernel follows a discrete-event (DE) MoC. This is similar to the MoC implemented as the kernels in Verilog and VHDL resulting in similar modeling styles and modeling problems. To our knowledge, the primary issue designers are faced with when using SystemC involves expressing concurrent behaviors correctly. This is a common difficulty when modeling designs in traditional HDLs that require concurrency and controlled resource sharing. This brings us to an interest in *pruning* [94] the way in which the concurrent designs are expressed. For example, design frameworks such as Ptolemy II [5] and languages such as SystemC-H [26] provide the designers with MoCs that exhibit concurrent behaviors such as the communicating sequential processes (CSP). Other frameworks such as SIGNAL [95] and Lustre [96] have been successfully used to describe highly concurrent applications. However, it is crucial that the pruning of nondeterminism also results in a model that can be further realized as an implementation model. Unfortunately, current integration of SystemC into various design flows requires re-implementation of the design once the initial design space exploration is complete. As we mentioned before, the difficulty with SystemC being accepted into the existing design flows is the path to an implementation model, partly due to the lagging behavioral synthesis tools for SystemC.

To counter some of these drawbacks in SystemC, we enlist the essential aspects that SystemC must provide:

1. a better method for describing concurrent designs,
2. a model that can be directly mapped to its implementation model and
3. a methodology to allow SystemC to fit current design flows.
4. appropriate interoperability techniques with SystemC reference implementation.

In this section, we propose a kernel-level extension for SystemC that implements Bluespec's rule-based MoC. A kernel-level extension is better suited than a language-level implementation (adding hierarchical and primitive channels to model the MoC using SystemC constructs) for simulation efficiency reasons [33, 97, 98]. In addition, this MoC has successfully shown its usefulness in easing the task of describing concurrent behaviors and there is a direct mapping from this rule-based MoC to a hardware implementation model. Our work here describes the rule-based MoC extension for SystemC that we call Bluespec-ESL (BS-ESL). An important aspect about introducing the rule-based MoC is to investigate how to correctly interoperate it with SystemC's DE. For that reason the second part of this section emphasizes on being able to co-simulate BS-ESL with SystemC.

6.1 Advantages of this Work

The advantages of this work is as follows:

Shared-state concurrency: When compared to the rule-based MoC, SystemC's in-built DE MoC does not adequately ease the complexity of shared state concurrency. This is because of SystemC's computation model that includes threads that are synchronized with events. The designer must explicitly manage the thread accesses to the shared states. Bluespec's rule-based MoC handles the shared state concurrency better with guarded atomic rules and we extend SystemC with this capability in this work.

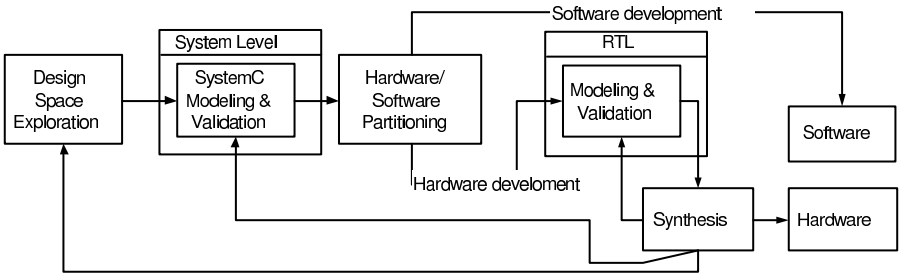
Co-simulation environment: Making BS-ESL interoperable with SystemC provides an environment where C/C++/SystemC can be used for software models and BS-ESL for hardware models. Therefore, the need for using application programming interfaces or programming language interfaces is avoided.

Reuse of IPs: There is a large code base of SystemC components distributed as IPs. These IPs can be effectively reused for hardware/software co-simulation even when users want to model with rule-based MoC because we provide the interoperability.

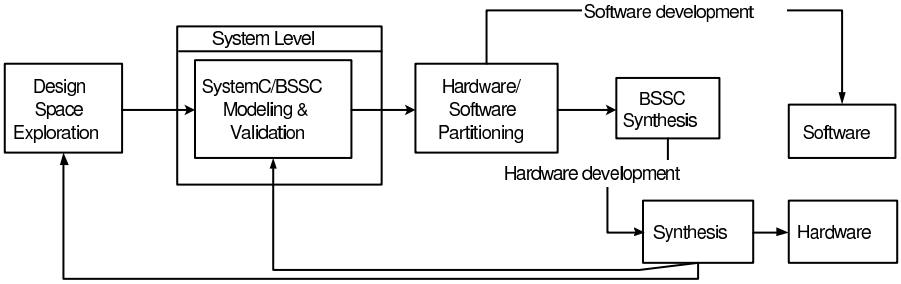
Synthesizability: Not all hardware descriptions in SystemC are guaranteed to be synthesizable, and the current high-level synthesis tools impose certain restrictions to which the designer must adhere to for synthesizability. On the other hand, Bluespec can generate synthesizable Verilog from the Bluespec MoC, hence providing a fully synthesizable hardware model within SystemC itself.

6.2 Design Flow

SystemC's main use in current industry design flows is for system-level modeling and simulation for fast design space exploration and validation of concepts as shown in Figure 6.1(a).



(a) Current Design Flow for SystemC



(b) Envisioned Design Flow with SystemC and BS-ESL

Figure 6.1: Before and After Design Flows

However, once the modeling and validation of the model is complete, the model undergoes hardware and software partitioning through which the software and hardware development take their own paths. For the most part the software is already modeled with SystemC/C++, but the hardware development requires re-implementation of the model at RTL with a synthesizable HDL. Using traditional synthesis tools, the hardware is generated. There are some cases that employ high-level synthesis tools such as Cynthesizer [90] to attempt synthesis from SystemC models. However, mainstream industries also continue with the legacy design flow of either implementing the model in Verilog or VHDL. In the case of Figure 6.1(b), we envision design space exploration using the rule-enhanced SystemC such that software components (new, or existing IP) are modeled using SystemC and the anticipated hardware using BS-ESL. After hardware/software partitioning, the design flow is similar to the one presented in Figure 6.1(a) except that BS-ESL synthesis generates the hardware components into synthesizable Verilog. Notice the crucial difference between the two design flows is that we rely on the proven fact that a synthesizable Verilog model can be generated from the Bluespec rule-based MoC. So the need for re-modeling the hardware design in an HDL is avoided.

Figure 7.3 shows the design process when using BS-ESL. The designer describes the model using the BS-ESL language in header files (.hpp) on which the BS-ESLPP executes. The BS-ESLPP is necessary for enabling SystemC with the Bluespec MoC because implementing atomicity requires non-local and context-sensitive information. The BS-ESLPP analyzes the source code to generate the registration calls for the kernel, the implicit conditions and

syntactic checks on the described model using BS-ESL. The preprocessor generates additional functions (.bpp) in separate files from the original source files (.hpp). In the next stage, the BS-ESL model is complied with the .bpp files with SystemC and BS-ESL runtime libraries and headers to generate the executable.

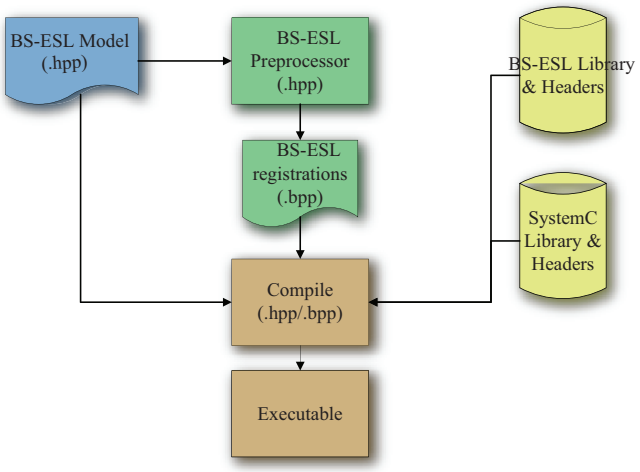


Figure 6.2: BS-ESL Design Flow

6.3 BS-ESL Language

The BS-ESL extensions include syntactic constructs for modules, interfaces, rules and methods. Following C++ and SystemC tradition, these extensions are defined as macros in a standard include file, bsystemc.h. These macros expand into calls to functions that are generated by the BS-ESLPP which, in turn, register various kinds of information with the rule-execution kernel.

In general, our approach uses class-based abstractions for modeling the constructs of BS-ESL. This means that we attempt to capture most constructs as classes, for example, BS-ESL modules, interfaces and nested interfaces. We provide several macros as BS-ESL constructs and mandate their use. Their expansion results in the appropriate C++. During our discussion on the language syntax, we describe the essential macros as well as the relevant expanded C++ resulting from it. In doing so, it will be evident how to employ the BS-ESL extension using plain C++ for designers who wish to avoid using macros.

6.3.1 BS-ESL Modules

The first construct we present is the definition of a module. The concept of a module in BS-ESL is the same as in SystemC and Bluespec where a design may consist of multiple modules. In fact, the BSSC module inherits from SystemC's `sc_module` class. Furthermore, modules allow for structural hierarchy such that a component modeled as one module may be embedded as a sub-component of another module. A module may contain rules, methods or interface implementations in addition to variable declarations and utility function definitions. It must also contain a module constructor.

The basic module definition macro is shown in [Listing 6.1, Line 1] that takes in the `name` and `prov_if` arguments. The `name` argument defines a module class using that parameter, whereas the `prov_if` is the argument of the providing interface. In addition, every module declaration must have its corresponding constructor. We require the use of the `ESL_CTOR()` macro shown in [Listing 6.1, Line 2] to indicate the start of the constructor and the `ESL_END_CTOR` ([Listing 6.1, Line 3]) at the end before closing the curly braces. `ESL_CTOR()` is a variadic macro allowing for more than two arguments as parameters to the macro. The first is the `name` of the class that must be the same as the name associated with the module class. The following arguments can be parameters for the constructor. Note that Bluespec uses the terminology *provides an interface* and *uses an interface* which are synonymous to *implements an interface* and *uses an interface* in C++, respectively.

Listing 6.1: Module and Constructor Macros

```
1 ESL_MODULE ( name, prov_if )
2 ESL_CTOR(name, args...)
3 ESL_END_CTOR
```

An example definition in [Listing 6.2, Line 1] shows that a module class is defined with the name `mkInitiator` and it implements an interface `MASTER_IFC`. Its corresponding macro expansion is shown in [Listing 6.2, Line 11 - 27]. Notice that before we actually start defining the `mkInitiator` class, we create a `typedef` class `__mkInitiator`. The `__mkInitiator` class contains two `typedef`s. They are `ESL_CURRENT_MODULE` and `ESL_CURRENT_INTERFACE`, where the former represents the type of the module class and the latter the type of the interface it provides. However, since the type of the module class is not yet known, we must precede the typedef class definition with a forward declaration of the module class `mkInitiator`. This enables the module class declaration to inherit this typedef class such that it has access to a partial type of itself and the type of the interface. This is used in other module and constructor macros. This indirection helps in solving some of the issues related to nested classes in C++ that we discuss in section 6.3.6. The declaration of `mkInitiator` has multiple inheritance relationship with the `sc_module` (the class that defines a SystemC module), `esl_module` (a BS-ESL specific module class), the typedef class and the interface.

We also show the module constructor in [Listing 6.2, Line 3 - 7] and its expansion. The `ESL_END_CTOR` is required to perform two tasks; firstly storing the `this` pointer to the `_esl_this` member of the base interface class `esl_if`. This is later used in referring to the module

class's instance for sub-interfaces. We describe this in a later section. In addition to that, we also add a member function prototype for `esl_registrations()`, which we invoke at the end of the constructor. This registration call is automatically generated using the BS-ESL preprocessor to register the BS-ESL modules into the BS-ESL runtime system.

Listing 6.2: Example Uses of Module Macros

```
1  ESL_MODULE ( mkInitiator, MASTER_IFC ) {
2    // ...
3    ESL_CTOR(mkInitiator, UId _id, int _incr): id(_id), incr(_incr)
4    {
5      //...
6      ESL_END_CTOR;
7    };
8  };
9
10 /* Expanded */
11 class mkInitiator;
12 struct __mkInitiator
13 {
14   typedef mkInitiator ESL_CURRENT_MODULE;
15   typedef MASTER_IFC ESL_CURRENT_INTERFACE;
16 };
17 class mkInitiator : public sc_module, public esl_module, public __mkInitiator,
       public MASTER_IFC
18 {
19   //...
20   void esl_registrations();
21   public:
22   mkInitiator (sc_module_name , UId _id, int _incr): id(_id), incr(_incr)
23   {
24     // ...
25     _esl_this = this; esl_registrations();
26   };
27 }
28 /* End Expansion */
```

6.3.2 BS-ESL Rules

The basic building blocks for rule-based MoCs are the rules themselves. In BS-ESL, rules are modeled as private member functions of the module's class. A rule has a unique name per module and it is a guard-action pair. The guards evaluate to Boolean values and upon a successful guard condition, its respective action is triggered. We show that a `ESL_RULE()` macro takes two arguments as its input in [Listing 6.3, Line 2]. The first is the name of the rule `rule_name` and the second is an expression that evaluates to a Boolean value as the `rule_guard`. The body of the `ESL_RULE()` macro describes the action associated with this rule. An example of a rule definition is shown in [Listing 6.3, Line 7]. The name of the rule is `countdown_a` and the guard is that the `control` data member cannot be zero and the `count` must be greater than one. If this guard evaluates to true, then the `count` data member is decremented.

As shown in the expansion of the macro in [Listing 6.3, Line 14], the rule construct is expanded into three member functions. `rule_countdown_a_implicit()` (whose full definition is generated by the BS-ESL preprocessor) encapsulates all the implicit conditions of all the methods invoked in the rule, `rule_countdown_a_RDY()` encapsulates the rule's explicit condition, so that it can be tested separately before invoking the rule body. Note that it also tests the implicit conditions. Finally, `rule_countdown_a()` just encapsulates the rule body, so that it can be invoked by the runtime executor.

Listing 6.3: BS-ESL Rules

```
1  /* Rule definition */
2  ESL_RULE (rule_name, rule_guard) {
3    // rule_action
4  };
5
6  /* Rule definition example */
7  ESL_RULE (countdown_a, (control != 0) && (count > 1)) {
8      count = count - 1;
9  }
10 /* Expansion */
11 bool rule_countdown_a_implicit ();
12 bool rule_countdown_a_RDY ()
13 {
14   return (((control != 0) && (count > 1)) && rule_countdown_a_implicit());
15 }
16 void rule_countdown_a () {
17   count = count - 1;
18 }
19 /* End expansion */
```

6.3.3 BS-ESL Methods

Bluespec has three kinds of methods available for use and they are action, value and actionvalue methods. The differences between these are that a value method only returns a value without altering any states in the module, an action method changes the state of the module and an actionvalue method changes the state as well as returning a value. In C++, all functions can change the state and return a value (though the return type may be `void`) so the error checking for the semantics associated with the different types of methods is handled in the BS-ESL preprocessor. The three method macros are variadic macros and they are shown in Listing 6.4. The `name` is the member function name, the guard is the explicit condition that the BS-ESL runtime system uses for scheduling purposes, the `return_type` is the return type for the value method and `args...` are the arguments the member function can have as function parameters.

Listing 6.4: BS-ESL Methods

```
1  ESL_METHOD_ACTION ( name, guard, args...)
2  {
3    //...
```

```
4  };
5  ESL_METHOD_ACTION_VALUE (name, guard, args...)
6  {
7    //...
8  }
9  ESL_METHOD_VALUE (name, guard, return_type, args...)
10 {
11   //...
12 }
```

We do not show the expansion for these macros because they are expanded in a similar fashion to the ones presented for BS-ESL rules. Each method yields three member functions; the first being the implicit condition member function, the second testing for the explicit condition (guard) and the third the implementation associated with the method.

6.3.4 Interfaces

The concept of interfaces is different in Bluespec than in C/C++ and SystemC. In the former case, every module must provide an interface, whereas in the latter a class or module does not necessarily have to implement an interface. Following Bluespec's approach, BS-ESL requires every module to provide an interface, but it may be of type `ESL_EMPTY`, which is a special data-type indicating that the particular module provides an empty interface (an interface with no methods).

The macros we provide for declaring BS-ESL interfaces are shown in Listing 6.5. The BS-ESL interface can contain action, value, and actionvalue interface methods. Each of these interface method macros expand into three pure virtual member function prototypes for the interface class named `if_name`. These three member functions are similar to those expanded for rules, but with interfaces these member functions are only prototypes without any implementation. A member function in C++ requires a return type, a function name, and its arguments. Hence, our macros for these methods shown in [Listing 6.5, Line 2 - 4] also require parameters for the `return_type`, `name` and `argument list` with the exception for the action method which expands to a `void` member function.

Listing 6.5: BS-ESL Interface Macros

```
1 ESL_INTERFACE ( if_name ) {
2   ESL_METHOD_ACTION_INTERFACE ( name [, argument list...]);
3   ESL_METHOD_VALUE_INTERFACE (return_type, name [, argument list...]);
4   ESL_METHOD_ACTIONVALUE_INTERFACE (return_type, name [, argument list...]);
5 };
```

Listing 6.6 presents an example of declaring an interface for an action method and its expansion. Once again, this is exactly the same scheme of expansion as used when expanding BS-ESL rules except that the expanded members are pure virtual function prototypes. In addition, the interface class inherits from `esl_if`, which is a BS-ESL base interface class. Every interface inherits from this base interface class.

Listing 6.6: BS-ESL Method Example

```
1  template <typename T>
2  ESL_INTERFACE( FIFO )
3  {
4    ESL_METHOD_ACTION_INTERFACE (enq, const T&);
5    //...
6  };
7  /* Expansion */
8  template <typename T>
9  struct FIFO:public esl_if
10 {
11   virtual void enq (const T&) = 0;
12   virtual bool method_enq_RDY () = 0;
13   virtual bool method_enq_implicit () = 0;
14 };
```

6.3.5 BS-ESL Primitive Modules

The BS-ESL extension provides the designer with certain BS-ESL specific primitive modules as well. They are `esl_reg`, `esl_wire` and `esl_rwire`. It must be noted that these primitive modules are different from any channels in SystemC. This is necessary because SystemC follows the discrete-event semantics which is different from the BS-ESL's rule-based semantics. The `esl_reg` primitive module models a basic register with which the BS-ESL runtime system ensures that in one cycle, all reads to the register precede the writes to it. The value on the register can be valid for multiple cycles. The `esl_wire` only stores the value for the current clock cycle and the BS-ESL runtime system schedules the rules such that all writes on a wire occur before reads. Similarly, `esl_rwire` is updated with the writes before the reads are allowed. However, an additional `isValid()` member is available in `esl_rwire` to allow for testing of whether the value is valid in the wire. These primitive modules follow the same semantics as in Bluespec [99]. Furthermore, each of these primitive modules are templated such that any type either native C++, SystemC or classes can be transmitted across them.

6.3.6 Higher Abstraction Mechanisms

Templating

We provide macros that support templated parameters for modules and interface methods. The first two arguments are the same as the basic module definition macro but the third argument specifies the type of the templated argument. An example usage of this is shown in [Listing 6.7, Line 1] where the template argument is simply a type variable `T`. In [Listing 6.7, Line 5 - 8], we show an example of declaring a templated interface.

Listing 6.7: Templated Macro Definitions

```
1 ESL_MODULE_TEMPLATE( mkInitiator, MASTER_IFC, T ) {
2   ...
3 };
4 /* Templated variant */
5 template <typename T>
6 ESL_INTERFACE ( EX_IF ) {
7   ...
8 };
```

Interface Abstractions: Nesting

Bluespec and SystemC both have a notion of nested interfaces. We describe Bluespec's notion of nested interfaces with an example. Suppose that two templated interfaces `Get<G>` and `Put<P>` with `G get()` and `put(P &)` methods are declared. Then, these two interfaces can be used to declare other interfaces such as a `Master< REQ, RSP > = (Get<REQ>, Put<RSP>)` and `Slave< REQ, RSP > = (Put<REQ>, Get<RSP>)` with specific types `REQ` and `RSP`. Hence, the `Get<G>` and `Put<P>` interfaces are nested within the `Master<>` and `Slave<>` interfaces. Now, we could declare another interface `DMA< REQ, RSP > = (Slave< REQ, RSP>, Master< REQ, RSP>)` so that the `Master<>` and `Slave<>` interfaces are nested within the `DMA<>` interface.

Listing 6.8: BS-ESL Sub-interface Macros

```
1 /* Nested Interfaces */
2 ESL_INTERFACE ( if_name ) {
3   ESL_SUBINTERFACE ( if_name, sub_if_name);
4 };
```

The same example of nested interfaces in SystemC uses abstract classes as interface classes and inheritance relationships to nest the interfaces. This would be such that the `Master<>` and `Slave<>` interface classes both inherit the `Get<G>` and `Put<P>` interfaces, which are themselves inherited by the `DMA<>` interface. This would cause a problem because the type signature of the `get()` method are the same in the `Master<>` and `Slave<>` interfaces. Therefore, using the traditional inheritance relationship for nested interfaces is insufficient in describing the nested interface abstractions of Bluespec. Our solution to this involves using nested classes in C++ to provide nested interfaces. This is a starkly different approach than the one used in SystemC.

We introduce the idea that an interface may contain nested interfaces as sub-interfaces ([Listing 6.8, Line 3]). Our use of nested classes to represent nested interfaces results in a few awkward issues, especially when compared to SystemC/C++ programming styles for nested interfaces. The first issue with using nesting classes is that the sub-interfaces do not have access to the outer interface's members. The second issue is that unlike in the case of multiple inheritance, the sub-interfaces require their instances be able to use the interface. We resolve the first issue by the `ESL_CURRENT_MODULE` typedef mentioned earlier when

discussing the module definitions. Sub-interfaces use a pointer `esl_this` that is a cast of the `ESL_CURRENT_MODULE` type to give sub-interfaces access to the outer interface's members. Solution to the second issue results in making instantiations of the nested interfaces using the `ESL_PROVIDE()` macro.

Listing 6.9: Nested Interfaces in BS-ESL

```
1  template <typename req, typename rsp>
2  ESL_INTERFACE( Client )
3  {
4    ESL_SUBINTERFACE( Get<req>, request );
5    ESL_SUBINTERFACE( Put<rsp>, response );
6  };
7  typedef Client<ReqT, RspT> MASTER_IFC;
8  ESL_INTERFACE( MASTER_WITH_INTR_IN_IFC )
9  {
10   ESL_SUBINTERFACE( MASTER_IFC,  mwi_master );
11   ESL_SUBINTERFACE( Put<bool>,   mwi_intr );
12 };
13 /* Expansion */
14 typedef Client<ReqT, RspT> MASTER_IFC;
15 struct MASTER_WITH_INTR_IN_IFC:public esl_if
16 {
17   MASTER_IFC * mwi_master;
18   Put<bool> * mwi_intr;
19 };
```

Listing 6.9 shows an example of defining an interface in BS-ESL with the `ESL_INTERFACE()` macro with two sub-interfaces. For the sub-interfaces, the expansion declares pointers to the interface types in [Listing 6.9, Line 17 & 18]. When a module uses a nested interface, the outer interface class is passed as one of the arguments. However, implementing this interface has additional macros suffixed with `_DEF()`. We show snippets of an example that implements the interface from Listing 6.9 in Listing 6.10. The module declaration in [Listing 6.10, Line 1] declares a module that provides the `MASTER_WITH_INTR_IN_IFC` interface. Skipping some of the intermediate code, the `ESL_SUBINTERFACE_DEF()` macro implements the `MASTER_IFC` interface that requires implementation of `get` and `put`, but we only present the `get`. We implement the `get` as a value method using the `ESL_METHOD_VALUE()` macro. Notice in [Listing 6.10, Line 12] the use of `esl_this`. This allows access to the outer module's members such as the FIFO. It uses the `_esl_this` member of the base interface class `esl_if` and casts it to the module's type, thus allowing access to members of the module.

Listing 6.10: Nested Interfaces in BS-ESL

```
1  ESL_MODULE( mkInitiator_with_intr_in, MASTER_WITH_INTR_IN_IFC )
2  {
3    FIFO<ReqT> *out;
4    // ...
5    ESL_SUBINTERFACE_DEF( MASTER_IFC, mwi_master )
6    {
7      ESL_SUBINTERFACE_DEF( Get<ReqT>, request )
8      {
9  ESL_METHOD_VALUE (get, true, ReqT)
```

```
10        {
11    ReqT r = esl_this->out->first();
12    esl_this->out->deq();
13    return r;
14  }
15  ESL_SUBINTERFACE_CTOR (mwi_master)
16  {
17    ESL_PROVIDE( request );
18    // ...
19  };
20  /* Expansion for sub-interface constructor only */
21  __mwi_master ()
22        {
23          request = new __request();
24    // ...
25        };
26  /* End Expansion  */
27      };
28      // ...
29    };
30
31    ESL_CTOR(mkInitiator_with_intr_in, UId _id, int _incr): id(_id), incr(_incr)
32    {
33      // ...
34      ESL_PROVIDE( mwi_master );
35      // ...
36      ESL_END_CTOR;
37    };
38    /* Expansion for sub-interface constructor only */
39    mkInitiator_with_intr_in (sc_module_name , UId _id, int _incr): id(_id), incr(
          _incr)
40    {
41      // ...
42      mwi_master = new __mwi_master();
43      // ...
44    };
45    /* End Expansion  */
46  };
```

Using `esl_this` we resolved the issue of providing access to the outer class's members. However, the other issue of requiring instantiations of the nested classes still remains. We tackle this by providing a sub-interface constructor as in [Listing 6.10, Line 15] that uses the `ESL_PROVIDE()` macro to simply instantiate the object.

Communication Abstraction: mkConnection

BS-ESL defines two templated abstract interfaces `Get<T>` and `Put<T>` and they implement the `get()` and `put()` methods. The `get()` method retrieves one token of data from a particular data structure and the `put()` adds a token of data into a data structure. These are interface methods such that the designer must implement them when using the interface. BS-ESL consists of some library components such as a `FIFO` that implements the `Get<T>`

and `Put<T>` interfaces. `FIFO`'s `get()` method implementation returns and removes the first token of data from a queue. Similarly the `put()` pushes one token of data onto the queue. Furthermore, we enforce the restriction that the `get()` can only be invoked when the `FIFO` is not empty and likewise the `put()` can be invoked only when the `FIFO` is not full.

In a transaction-level model, or any higher lever of abstraction, connecting modules using `FIFO`s is very common. So, we provide `mkConnection` for connecting a `Get<T>` interface to a `Put<T>` interface. The mechanism is a templated module whose constructor takes as arguments one module's `Get<T>` interface, and another module's `Put<T>` interface. `mkConnection` has a single rule that calls the `get()` and `put()` methods of the respective modules. The rule will not fire if there is no data to transfer or if the data cannot be transferred.

`mkConnection` makes it simple to connect modules together. Especially at a higher level of abstraction that uses a master/slave or request/response interconnection. Instead of connecting modules hierarchically, they are connected through `mkConnection`. Furthermore, if it is observed that the same pair of request/response and get/put interfaces are used in many places. The interfaces themselves can be combined to form a single nested interface consisting of a `Get<T>` and `Put<T>` interface. We call this the `Client` interface with its dual being the `Server` interface. With builtin `Get<T>`, `Put<T>`, `Client`, and `Server` interfaces, and two versions of `mkConnection`, modules at a transaction level of abstraction can be connected easily, and the connection is still realizable in hardware.

6.4 BS-ESL Execution

As usual in SystemC, the function `sc_main()` should contain statements that instantiate the module hierarchy, e.g., by instantiating an object of the top-level `ESL_MODULE()` class. As the module hierarchy is instantiated, the constructors perform certain registrations with the BS-ESL runtime system. Finally, a call to `sc_start()` begins execution.

The BS-ESL runtime system is an additional library linked in with the standard SystemC library. It consists of a rule scheduler and a rule execution kernel. When execution begins, as a consequence of `sc_start()`, the scheduler runs once to analyze all the rules and rule-based interface methods, and produces a schedule indicating which rules can execute simultaneously within a clock cycle. The rule execution kernel then repeatedly executes rules in each clock according to this schedule.

SystemC has an optimization option whereby some threads can be declared as `SC_METHOD`s instead of `SC_THREAD`s; such threads can never suspend, and so can be executed within the context of an invoking thread. Similarly, the BS-ESL execution kernel has an optimization option called `esl_run()` whereby the programmer can exercise finer control over rule execution.

6.5 An Example Demonstrating BS-ESL and SystemC Integration

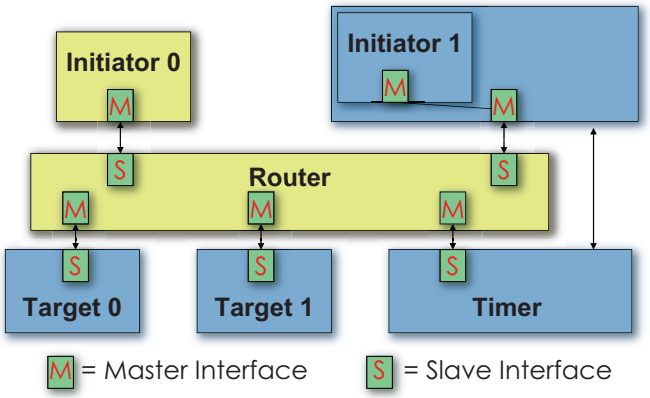


Figure 6.3: Transaction Accurate Communication (TAC) Example

Initially the design presented in Figure 6.3 uses core SystemC modules and standard TLM `get()` and `put()` interfaces [100]. However, we replace the router, some initiators and some targets to investigate hardware accuracy of these modules by implementing them using BS-ESL. We connect the SystemC modules with the BS-ESL modules via `mkConnection`. This allows connecting any combination of core SystemC, BS-ESL `get()` and `put()` and master/slave connections through the overloading of `mkConnection`. Simulation of the model occurs through the standard `sc_start()` function. However, for fine-grained control over the simulation the `esl_run()` function may be used to simulate the BS-ESL model for a specified number of cycles.

6.6 Summary

This work presents BS-ESL, our kernel-level extension of Bluespec's rule-based MoC for SystemC. We employ a macro-based approach similar to that of SystemC to create the BS-ESL language using libraries and macros alone. Therefore, we can generate simulation models using C++ compilers just like SystemC. These macros are essential in correctly describing modules, rules, interfaces, and nested interfaces. They are modeled such that they are semantically the same as the constructs in the rule-based MoC described in Bluespec SystemVerilog. Bluespec compiler can generate synthesizable Verilog from their Bluespec SystemVerilog description. Leveraging this, we described how BS-ESL fits into current design flows by not only providing an interoperable rule-based MoC for SystemC, but also a path through which hardware synthesis is possible directly from BS-ESL descriptions. Such

a synthesis tool has been proposed by Bluespec.

In addition to the macros, we also provide BS-ESLPP, a BS-ESL preprocessor that is responsible for extracting context-sensitive information about the model to automatically generate registration calls for the BS-ESL scheduling and execution kernel. Our experience suggests that a difficulty in modeling with BS-ESL is the lack of debugging support. This is particularly troublesome because of the complex macro expansions. In order to better aid designer's modeling efficiency, we to extend BS-ESLPP to perform certain syntax and semantic checks that may be difficult to decipher from the macro expansions. We also plan to incorporate options into BS-ESLPP that automatically add debugging structures into the model that enable better debugging and logging of model execution.

6.7 Interoperability between SystemC and BS-ESL

Bluespec-SystemC (BS-ESL) extension provides designers with an MoC that simplifies the task of describing concurrent behaviors based on the notion of "atomic" rules in the rule-based MoC of Bluespec [101, 39, 41, 99]. Moreover, Bluespec has tools for synthesis from its Bluespec-SystemVerilog descriptions to Verilog [39, 41, 40] and a synthesis path from BS-ESL to Verilog. Both SystemC and BS-ESL can effectively model RTL. However, the manner in which designs are simulated are different because they embody distinct MoCs. SystemC employs delta cycles, delta-events, and events for simulating its designs. BS-ESL creates a topologically sorted schedule of its rules for simulating its designs. The simulation semantics of the two languages are different because even when BS-ESL is modeling RTL, it maintains an operational abstraction over HDL based RTL. This gives rise to an impedance mismatch or "simulation semantic gap" if two models, one in BS-ESL and one in SystemC are connected directly. We show that this mismatch leads to incorrect simulation behavior when a BS-ESL model is combinationally connected to a model using an HDL RTL such as SystemC. In this section, we use the acronym HDL RTL to mean any discrete-event (DE) simulation based HDL such as VHDL, Verilog or SystemC. Our approach in investigating the simulation semantic gap leads us to formalizing the SystemC and BS-ESL's simulation semantics.

In [47, 45, 102], formal operational semantics of SystemC simulation in terms of Abstract State Machines (ASM) [79, 78] have been presented. However, to make the issue of interoperability between the two MoCs considered here, we do not need to introduce ASM or ASM based semantics, as those will unduly burden the understandability of the basic issues under discourse. Instead, we provide a simple hypergraph based semantics for discrete-event simulation of SystemC RTL models. The simplicity of this model helps make the problem clearer, and our solution almost instantly understandable. We basically identify the problem in composing SystemC and BS-ESL models and propose techniques for resolving it, with proper correctness arguments.

The examples in this section are simple to illustrate the basic issues, but our interoperability semantics has been employed in BS-ESL and works for much larger models. To our knowledge, the interoperability issue between Bluespec's rule-based MoC and SystemC's DE MoC has not been addressed before this. Although described in the context of SystemC RTL, our interoperability semantics holds for simulating traditional HDLs against Bluespec as well. For this reason, we restrict our analysis to only RTL models. However, these techniques are applicable to a certain subset of TL models (TLM) and not all, but this is beyond the scope of our discourse here.

6.7.1 Problem Statement: Interoperability between SystemC and BS-ESL

We extended SystemC with a rule-based MoC extension called BS-ESL described earlier in this section [103]. We leverage that work and focus on the aspect of integrating an RTL design described in SystemC (or any HDL) with an RTL design described using BS-ESL. Due to the distinct MoCs that SystemC and BS-ESL follow, there are issues in interoperability that need to be carefully investigated. We do this in this section. Our approach formally presents the interoperability problem and provides a wrapper based solution on this analysis.

6.7.2 Interoperability Issues at RTL

Note that SystemC simulation follows DE [8] semantics, which does not mandate any specified order of execution of the SystemC processes. As already mentioned, BS-ESL follows the rule-based semantics with a topologically sorted order of rules for execution. So, two designs at RTL will simulate differently. To provide a brief overview of this, take Figure 6.4 as an example of a simple circuit. For illustration purposes, suppose that all three components were modeled as SystemC processes. The order in which any of the processes are triggered does not impact the final result `y` because *delta-events* [8] are used for changes that must be processed within a cycle. Therefore, the state elements in Figure 6.4 receive the correct values. Now, suppose that component `C` is modeled using BS-ESL and the firing order of the processes is `A`, `C` then `B`. `A` generates one input which causes `C` to fire. This invokes BS-ESL's runtime executor that fires the rules in the topologically sorted order and updates the state element at the end of the iteration. The result in the state element contains incorrect values because the correct input from `B` is yet to arrive. So, when `B` executes, `C` will refire using the incorrect value in the state element. This occurs because delta-events are not employed in BS-ESL's semantics. This mismatch is a result of the semantic gap between SystemC and BS-ESL.

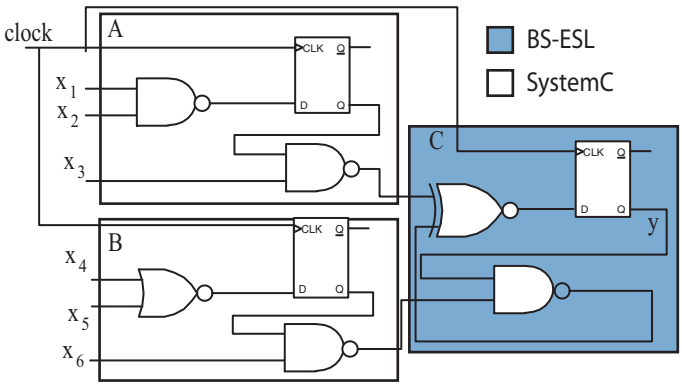


Figure 6.4: Simple Example of SystemC & BS-ESL

6.7.3 Interoperability Issues at TL

We present another example of a `GCD` calculator that illustrates the issue of mismatched simulation semantics with TLM. Note that only the `GCD` and `Display` components are clocked and the transfer between the components is via transactions; hence modeled at the cycle-accurate TL abstraction as per the OSCI TLM [8] standards. Figure 6.5 shows two input generators that act as external data entry points arriving at any time. The `GCD` calculator contains three internal registers to store the two inputs and the result. It performs an iteration of BS-ESL's runtime executor on every clock edge and writes the result when an appropriate condition is met. This register is read by the `Display` component. If the input generators produce new values within the same clock cycle after `GCD` has already been triggered then the computation must be redone with the "latest" input values. Otherwise the registers will hold incorrect values. Furthermore, BS-ESL does not have a notion or interface for delta-events so a direct composition is not possible. We follow this informal description of the issues with a formal presentation in the remainder of the section focusing only on the RTL scenario.

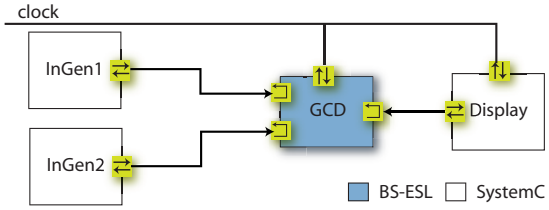
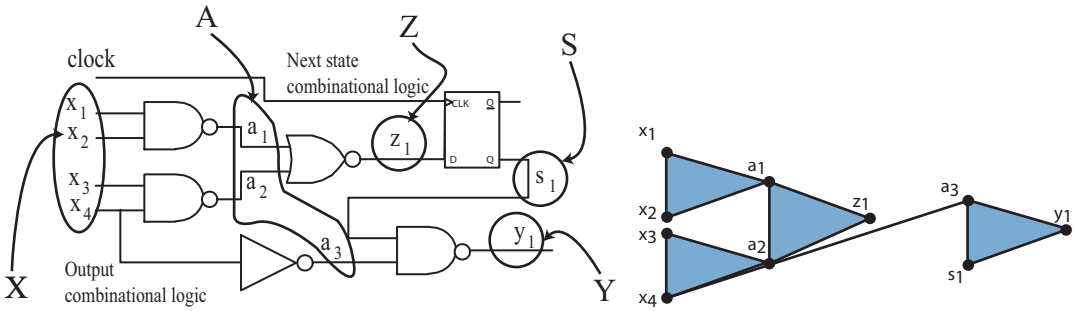


Figure 6.5: GCD Example at TL

6.8 Problem Description

Now that we have informally discussed the problem of interoperability between SystemC and BS-ESL. We present a formal treatment of the problem. We only provide few of the correctness proofs for the stated theorems. This is because most of the theorems can be easily shown to be correct. So, we begin by formalizing the notion of a simulation behavior, which is the snapshot of register values at the beginning of each clock cycle (Definition 6).

Definition 6. *Simulation behavior S_b of a model M at RTL level is defined as a function as follows: $S_b : N \rightarrow S \rightarrow V$, where $N = \{0, 1, 2, ...\}$ denotes the sequence of clock cycles, S is the set of registers (states) and V is the domain of values from which the registers take values. We use S_b^M to denote the simulation behavior of a model M.*



(a) Circuit (b) Hypergraph

Figure 6.6: Example Circuit

6.8.1 DE Simulation Semantics for RTL

This section formalizes an RTL circuit description and how current DE simulation semantics employ the use of delta cycles to compute correct outputs. We also formally present Bluespec's BS-ESL MoC and show how direct composition of the two may not result in the expected simulation behavior. During our formal presentation we slightly abuse notations and write $\hat{x} \in X$ to mean $\hat{x} \in X^k$ for some k and $\hat{u} \in VARS \backslash Y \backslash Z$ to mean $\hat{u}$ is a vector of variables where each element belongs to $VARS \backslash Y \backslash Z$. Note that our description of the DE simulation semantics considers a combinational cycle as an incorrect description, thus we disallow cyclic circuits. Other works suggest the benefits of cyclic circuits [104, 105, 106], but the treatment of cyclic circuits is beyond the scope of this discourse. Additionally, false loops [107, 108] are also not explicitly discussed in this section. This is because SystemC itself allows designers to intentionally model false loops and thus it is possible to create an interoperable model with BS-ESL that also contains a false loop. However, usually false loops result from resource sharing during logic synthesis and usually RTL designs do not contain them.

Let

$X = \{x_1, x_2, ..., x_k\}$	be the set of input signals
$Y = \{y_1, y_2, ..., y_l\}$	be the set of output signals
$A = \{a_1, a_2, ..., a_p\}$	be the set of intermediate combinational signals
$Z = \{z_1, z_2, ..., z_n\}$	be the set of internal combinational signals input to the state elements
$S = \{s_1, s_2, ..., s_n\}$	be the set of signals representing state element values
$VARS = X \cup A \cup Z \cup Y \cup S$	be the set of all variables

These sets are shown for the circuit in Figure 6.6(a). The set of equations that yield output

on the signals labeled Z, Y and A are produced by the combinational logic in Figure 6.6(a). We represent these as F_c^z, F_c^y and F_c^a respectively. Then, F_c represents the combinational logic of the entire circuit as a set of equations, $F_c = \{F_c^z \cup F_c^y \cup F_c^a\}$ where

$$\begin{aligned} F_c^z &= \{< z_m = f_m^z(\hat{u}) > \mid m = 1, 2, ..., n\} \\ F_c^y &= \{< y_j = f_j^y(\hat{v}) > \mid j = 1, 2, ..., l\} \\ F_c^a &= \{< a_i = f_i^a(\hat{w}) > \mid i = 1, 2, ..., p \\ &\qquad \text{and } \hat{w} \text{ does not contain } a_i\} \end{aligned}$$

and $\hat{u}$, $\hat{v}$, $\hat{w}$ are vectors of variables over the set $(VARS \backslash Y) \backslash Z$. The next state assignments denoted by S in Figure 6.6(a) are formalized by a set of delayed assignments $F_s = \{< s_k \Leftarrow z_k > \mid k = 1, 2, ..., n\}$. Delayed assignments update the value in the state elements at the beginning of the next clock cycle or at the end of the current cycle.

In hardware, signals propagate from input and current state through the combinational logic within one clock cycle and produce the outputs and latch the next state onto the state elements. Figure 6.6(a) describes an RTL circuit. However, simulating the combinational part amounts to a number of function computations, which in RTL description are not necessarily described in their topological order in the actual circuit. Therefore, most DE simulators need to reevaluate the functions several times using delta steps before all the combinational values are correctly computed.

We formalize any RTL design as a tuple $M =< X, Y, A, Z, S, F_s, F_c >$. F_c syntactically represents all the computation that happens during one clock cycle in the combinational part of the circuit. F_s captures the change of state at the end of a clock cycle when combinational computation and signal propagation are over. One can capture the execution semantics of a static circuit description $< X, Y, A, Z, S, F_s, F_c >$ using an acyclic hypergraph model. A hypergraph is a graph in which generalized edges may connect more than two vertices. So a hypergraph $G = (V, E)$ has $E \subseteq \bigcup_{k=1}^{|V|} V^k$, which means an edge may connect up to $|V|$ vertices. Figure 6.6(b) shows the corresponding hypergraph representation for the example circuit.

The way to visualize the execution semantics in Figure 6.6(b) of one cycle worth of combinational computation of an RTL design is as follows: consider a directed hypergraph where a hyperedge of the form $< a, b, c, d, e >$ would mean the nodes a, b, c, d together are all connected to node e. So $G_H = (V, E)$ where $V = X \cup Y \cup A \cup Z \cup S$ (variables from S can only be used as inputs) and

$$\begin{aligned} < \hat{u}, z_m > \in E \quad &\text{iff } < z_m = f_m^z(\hat{u}) > \in F_c \\ < \hat{v}, y_j > \in E \quad &\text{iff } < y_j = f_j^y(\hat{v}) > \in F_c \\ < \hat{w}, a_i > \in E \quad &\text{iff } < a_i = f_i^a(\hat{w}) > \in F_c \end{aligned}$$

In G_H, each hyperedge is marked with the equational assignment in F_c that gave rise to the hyperedge. The following observations are obvious:

Observation 6.8.1. *(1) G_H is an acyclic hypergraph for combinational loop-free RTL designs, (2) G_H can be topologically sorted, and (3) for simulation purposes, if we evaluate the variables in their topological sorted order in G_H, we obtain the correct values without having to reevaluate any of the nodes.*

Figure 6.6(b) shows a hypergraph representation of the circuit description in Figure 6.6(a). Notice that $s1$ is used only as an input to the output $y1$. From Figure 6.6(b) it is obvious that a circuit such as this can be topologically sorted. However, most RTL simulators, including SystemC do not topologically sort the design and therefore, notions of delta cycles and delta-events are used to get the same effect. They achieve the same effect as can be shown by proving the following theorem.

Definition 6.8.2. *Let a model M_1 simulated using delta cycle semantics have the simulation behavior $S_b^{M_1}$. Let the same model evaluated by topologically ordering the hypergraph be M_2 and has the simulation behavior $S_b^{M_2}$. Then, $\forall s_j \in S, \forall i \in N$, $(S_b^{M_1} i)s_j = (S_b^{M_2} i)s_j$.*

Proof. This proof is by induction on i. For $i = 1$, $\forall s_j \in S, (S_b^{M_1} 1)s_j = (S_b^{M_2} 1)s_j$ can be proven by considering that before cycle 1, all state elements in S are initialized to the same values in both M_1 and M_2. The same inputs are provided in the beginning of cycle 1. Then the delta cycle based evaluation basically recomputes these equations in F_c whose target signals evaluated before the equations are earlier in the topological order. Hence, the eventual result of the evaluation is the same as that of the topological order. Assume that $i = n$, $\forall s_j \in S, (S_b^{M_1} n)s_j = (S_b^{M_2} n)s_j$ is true. Show that when $i = n + 1$, $\forall s_j \in S, (S_b^{M_1}(n+1))s_j = (S_b^{M_2}(n+1))s_j$. Note that when $i = n + 1$, we can use the same argument used for the base case again for the inductive reasoning. □

Even though DE simulators have delta cycles, they do not go ad infinitum when the combinational logic is correctly designed because delta cycles impose a fixed point computation [109]. To show this formally, consider $< x_1, x_2, ..., x_k, s_1, s_2, ...s_n >$ as a vector of variables that are immutable during simulation of a single cycle. Let $< a_1, a_2, ...a_p, z_1, z_2, ..., z_t, y_1, y_2, ...y_l >$ be variables which can possibly change several times during successive delta cycles. Let their initial values during the beginning of the evaluation process be *don't cares*, which means their values are arbitrary.

Now reconsider graph G_H with the variables (nodes) in the graph being annotated with their order in the topological sort. So variables in $(VARS \backslash X) \backslash S$ can be sorted in the topological order and we rename them as

$$w_1, w_2, ..., w_p, w_{p+1}, ..., w_{p+t}, w_{p+t+1}, ..., w_{p+t+l}$$

The following observation easily follows:

Observation 6.8.3. *If G_H is topologically ordered before simulation then, (1) once the equation of the form $w_1 = f_1(\hat{u})$ is evaluated, it no longer needs reevaluation and (2) once $w_j = f_j(\hat{u})$ is evaluated and $\forall i < j$ and $w_i = f_i(\hat{u})$ has already been evaluated, w_j does not need reevaluation.*

By Theorem 6.8.2 and Observation 6.8.3 it follows that even with arbitrary order of evaluation, the process of delta cycle terminates when no w_k is needed to be reevaluated. At this point, the delayed assignments in F_s can be evaluated and that ends one simulation cycle. This shows that delta cycles actually amount to a fix point computation for $< a_1, a_2, ..., a_p, z_1, z_2, ..., z_t, y_1, y_2, ...y_l >$ where applying one equation of F_c changes only one variable. This change in variable leads us to a new valuation, which is better (in an information order) than the old valuation or the same if it is the final correct valuation.

The use of delta-events is important for correctly simulating hardware in DE-based simulators. However, BS-ESL's rule-based MoC semantics simulate their designs differently, without the need of delta-events. We describe this next.

6.8.2 BS-ESL's Rule-based Semantics

A BS-ESL model is described as a 2-tuple $M = < V, R >$ of variables where V is the set of variables and R a set of rules. The rules are guarded commands of the form $g(\Gamma) \rightarrow b(\Lambda)$ where $\Gamma, \Lambda \subseteq V$, $g(\Gamma)$ is a Boolean function on the variables of Γ and $b(\Lambda)$ is a set of equations of the form $x_i = f_i(\Lambda)$ such that x_i is assigned the result of computing the value based on current values of variables in Λ. Using the definitions presented earlier, the variable set can be partitioned into four disjoint sets $V = X \cup Y \cup A \cup S$. Now, each rule $r_i \in R$ computes $b_i(\Gamma)$ and computes a number of functions $\{v_{ij} = f_{ij}(\hat{w}) \mid v_{ij} \in \Lambda \ \ and \ \hat{w} \in (VARS \backslash Y) \backslash Z\}$ where v_{ij} could be any variable. Now, we denote a syntactic model M_E equivalent to M, where $M_E =< X, Y, A, S, F_c >$. This means there is no distinction between Z and S and there is no F_s. This is because Bluespec's MoC topologically sorts the rules based on usage and assignment of variables, and since we disallow combinational loops, the variable dependency graph is acyclic; hence topologically sortable. As a result, the state variables are assigned once and only once, and there is no delta cycle based reassignment of these state variables. Therefore, the simulation semantics of Bluespec's rule-based MoC is different with respect to HDL RTL simulation semantics.

6.8.3 Composing HDL RTL Models

Let M_1 and M_2 be two RTL models defined as

$$\begin{aligned} M_1 = & < X_1, Y_1, A_1, Z_1, S_1, F_S^1, F_c^1 > \\ M_2 = & < X_2, Y_2, A_2, Z_2, S_2, F_S^2, F_c^2 > \end{aligned}$$

Note that M_1 and M_2 are both HDL RTL.

We compose them as $M_1 \oplus M_2$ in the simulation model such that $M_1 \oplus M_2 = < X, Y, A, Z, S, F_s, F_c >$. The way the composition works is that a subset of input variables of one module are fed by some of the output variables of the other module and vice versa. So $X_1 \cap Y_2$ represents the set of all output variables of M_2 that are fed into input of M_1 and $X_2 \cap Y_1$ represents the set of all inputs in M_2 fed by outputs of Y_1. Of course, to do this correctly, this should not result in any combinational loop. Also, $S_1 \cap X_2$ represents the set of state elements from M_1 feeding into M_2's inputs and likewise $S_2 \cap X_1$ feeds the inputs for M_1. So,

$$\begin{aligned} X = &\ (X_1 \cup X_2) - ((X_1 \cap Y_2) \cup (X_2 \cap Y_1) \\ &\ \cup (S_1 \cap X_2) \cup (S_2 \cap X_1)) \\ Y = &\ (Y_1 \cup Y_2) - ((X_1 \cap Y_2) \cup (X_2 \cap Y_1)) \\ Z = &\ Z_1 \cup Z_2 \\ A = &\ A_1 \cup A_2 \cup ((X_1 \cap Y_2) \cup (X_2 \cap Y_1)) \\ S = &\ S_1 \cup S_2 \\ F_s = &\ F_s^1 \cup F_s^2 \\ F_c = &\ F_c^1 \cup F_c^2 \end{aligned}$$

Note the following:

Observation 6.8.4. *A correct composition $M_1 \oplus M_2$ circuit has no combinational loops.*

Let M_1, M_2 both be RTL models or M_1, M_2 both be BS-ESL models. There are two cases covered in the next theorem.

Definition 6.8.5. *Let the composed model $M = M_1 \oplus M_2$ be (1) simulated using delta cycle simulators and the simulation behavior is $S_{b\Delta}^{M_1 \oplus M_2} : N \rightarrow S_1 \cup S_2 \rightarrow V$. (2) simulated using BS-ESL then $S_{b*}^{M_1 \oplus M_2} : N \rightarrow S_1 \cup S_2 \rightarrow V$. Then $\forall s_j \in S_1 \cup S_2, \forall i \in N, \forall \tau \in \{\Delta, *\}$*

$$(S_{b\tau}^{M_1 \oplus M_2} i)s_j = \begin{cases} (S_{b\tau}^{M_1} i)s_j & \text{if } s_j \in S_1 \\ (S_{b\tau}^{M_2} i)s_j & \text{if } s_j \in S_2 \\ (S_{b\tau}^{M_1} i)s_j = (S_{b\tau}^{M_2} i)s_j & \text{if } s_j \in S_1 \cap S_2 \end{cases}$$

Theorem 6.8.5 states that the simulation of M is correct when M_1 and M_2 are both modeled using delta cycle semantics and simulated using a delta cycle simulator. Likewise, the simulation of M is correct if both models are modeled using rule-based semantics and simulated using the rule-based MoC.

6.8.4 Composing HDL RTL & BS-ESL Models

Now we consider composing two models where M_1 is in HDL RTL and M_2 in BS-ESL. We show that the inherent MoCs of these two SLDLs do not allow for direct composition of the models.

$$\begin{aligned} M_1 = &< X_1, Y_1, A_1, Z_1, S_1, F_S^1, F_c^1 > \\ M_2 = &< V_2, R_2 > \\ &\text{such that } V_2 = X_2 \cup Y_2 \cup A_2 \cup S_2. \end{aligned}$$

Suppose we create a composition $M = M_1 \oplus M_2$ and ensure there are no combinational cycles as shown in the simple example in Figure 6.4. Then, M_2 is simulated using BS-ESL simulation semantics by topologically sorting its rules, and M_1 is simulated using any HDL's simulation semantics in DE with delta cycles. So, we claim that the result of simulation for M may differ from the correct result of simulating them if M_1 was first topologically sorted and then composed.

Definition 6.8.6. *If M_1 and M_2 has combinational path between them and M_1 is simulated using delta cycles, M_2 with rule-based execution then it may not be the case that $\forall i \in N, \forall s_j \in S_2$, $(S_b^{M_1 \oplus M_2} i)s_j = (S_b^{M_2} i)s_j$.*

As stated before, since $M_2 =< V_2, R_2 >$ corresponds to an execution model $M_{2E} =< X_2, Y_2, A_2, S_2, F_c^2 >$ where F_c^2 contains all assignments to variables including those in S_2. So, $M_1 \oplus M_2$ composition and topologically sorting the corresponding hypergraph will lead to $G_{H(M_1 \oplus M_{2E})}$. Figure 6.4 shows an example of a composed model. Unfortunately, now we can no longer guarantee that delta cycle based evaluation of $M_1 \oplus M_2$ will have the same values as hypergraph based evaluation. The reason is simple to see. If there is combinational path from M_1 and M_2, and during simulation M_2 is evaluated first based on rule-based semantics, the values of its state variables will change permanently for that cycle. This would mean that during reevaluation of M_1, if delta-events trigger recomputation of M_2, then the state variables would contribute their new values and not the values from the previous cycles.

Proof. We know that BS-ESL's simulation does not perform delta cycle based simulation. Thus, at the end of one evaluation of M_2, all state elements are assigned their computed values. In other words, in the RTL model described using BS-ESL, F_c and F_s are not distinguished. Now, suppose variable $x_2 \in X_2$ in M_2 is connected to output $y_1 \in Y_1$ in M_1. Since M_1 goes through delta cycles, y_1 may be re-assigned a few times, and every time it is re-assigned, M_2 must be reevaluated. But, this reevaluation will then be using the values of variables in S_2 that have been newly assigned during its last evaluation of M_2, which is not what happens in the state elements in M_1. Similarly, a reverse situation can be outlined. □

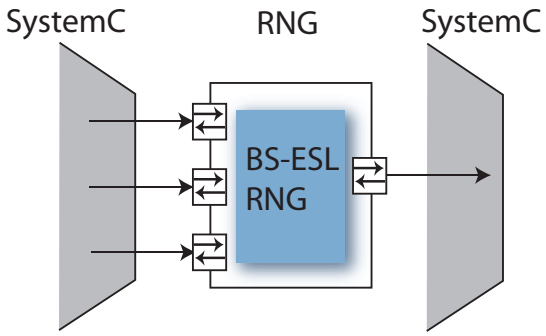


Figure 6.7: Block Diagram for RNG

We use Figure 6.7 and Table 6.1 to show the execution trace of a composition of an HDL RTL and BS-ESL model for a random number generator (RNG) example. Note that the BS-ESL takes three inputs from a SystemC design and one output to another part of the SystemC design. The three input ports are the clock, two integer inputs and the output is a result. This is described in the table as well. The vector notation describes values for $< p_1, p_2, P1_{Reg}, P2_{Reg}, result_{Reg} >$, where p_1 and p_2 are input signals, variables subscripted with Reg are registers internal to the component, $k * \Delta$ represents the k^{th} delta-event and $t+1$ represents the first cycle after time t. The seed for the RNG is specified as the initial value for $result_{Reg}$ as 79 and the random numbers are computed using $result_{Reg} := (P1_{Reg} * result_{Reg}) + P2_{Reg}$. For the HDL RTL case, the inputs arrive at different delta-events that are stored in their respective registers in the following clock cycle. From there on, the RNG computation iterates and stores the result. Similarly, the inputs arrive in delta cycles to the composed situation except that due to BS-ESL's ordered schedule and lack of delta-events, registers are updated at the end of the iteration. Hence, the $result_{Reg}$ contains an incorrect value which is continuously used for subsequent random numbers. This is incorrect behavior for a RNG because for the same seed and constants $p1$ and $p2$, the sequence of random numbers generated must be the same.

Table 6.1: Execution Trace for RNG

Sim. Time	HDL RTL	RTL & BS-ESL
t	$< 0, 0, 0, 0, 79 >$	$< 0, 0, 0, 0, 79 >$
$t + \Delta$	$< 263, 2, 0, 79 >$	$< 263, 2, 0, 0, 0 >$
$t + 2 * \Delta$	$< 263, 71, 0, 0, 79 >$	$< 263, 71, 263, 2, 2 >$
$t + 1$	$< 263, 71, 263, 71, 48 >$	$< 263, 71, 263, 71, 97 >$
$t + 2$	$< 263, 71, 263, 71, 95 >$	$< 263, 71, 263, 71, 82 >$
$t + 3$	$< 263, 71, 263, 71, 56 >$	$< 263, 71, 263, 71, 37 >$

6.9 Solution: Our Interoperability Technique

Having identified that a simple minded composition of HDL RTL (M_1) and BS-ESL (M_2) is not correct, we use a technique that enables correct composition.

6.9.1 HDL RTL with BS-ESL RTL

Suppose M_1 and M_2 where $M_1 =< X_1, Y_1, A_1, Z_1, S_1, F_S^1, F_c^1 >$ is an HDL RTL model and $M_2 =< V_2, R_2 >$ such that V_2 is described by $< X_2, Y_2, A_2, S_2, F_c^2 >$ is an BS-ESL model. Now, we must introduce a wrapper such that the composition $M_1 \oplus w(M_2)$ is possible. The wrapper is defined as
$w(X_2, Y_2, A_2, S_2, F_c^2) =< X^w, Y^w, A^w, Z^w, S^w, F_s^w, F_c^w >$ where

$$\begin{aligned} X^w &= X_2 \\ Y^w &= Y_2 \\ A^w &= A_2 \\ S^w &= S_2 \\ Z^w &= \{z_i | s_i \in S_2\} \\ F_s^w &= \{< s_i \Leftarrow z_i | z_i \in Z^w >\} \\ F_c^w &= \text{ with each occurrence of } s_i \text{ replaced by } z_i \end{aligned}$$

The wrapper introduces intermediate combinational signals Z^w that take inputs from A^w and serve as inputs to the state elements S^w. This is indicative of adding temporary variables in M_2 whose inputs are signals from Z^w and the outputs are to the state elements. At every execution of M_2 the result of the combinational circuit is assigned to the temporary variables. The state elements are then assigned values of z_i as per delayed assignment semantics. In addition, when there is a combinational path between the models $(X_1 \cap Y^w) \cup (X^w \cap Y_1)$, the updates on the signals must occur in delta cycles. Once again, since M_2 does not have the notion of delta cycles, we execute M_2 at every value change on its combinational inputs. The simulation is correct because only the temporary variables are updated with these executions and not the state elements. The state elements are then only updated at the beginning of the next clock cycle so as to retain correct simulation behavior.

The simulation behavior for M_1 is $S_b^{M_1}$ and M_2 is $S_b^{M_2}$ as per Definition 6. Then, using our wrapper we can avoid the problem presented in Theorem 6.8.6.

Definition 6.9.1. $\forall i \in N, \forall s_j \in S_1 \cup S_2$

$$(S_b^{M_1 \oplus w(M_2)} i) s_j = \begin{cases} (S_b^{M_1} i) s_j & \text{if } s_j \in S_1 \\ (S_b^{M_2} i) s_j & \text{if } s_j \in S_2 \\ (S_b^{M_1} i) s_j = (S_b^{w(M_2)} i) s_j & \text{if } s_j \in S_1 \cap S_2 \end{cases}$$

Our implementation uses a SystemC process as the wrapper process. This wrapper process interacts with other external SystemC processes and handles conversions from SystemC channels to the BS-ESL interfaces. For the wrapper process to trigger on every delta-event, we introduce Δ-channels. These channels internally employ the *sc_event_queue* in SystemC to notify events, but we allow for transferring data along with every event generated as well. We make the wrapper process sensitive to these Δ-channels such that whenever there is an event on the channels, the wrapper process is scheduled for execution within the same clock cycle.

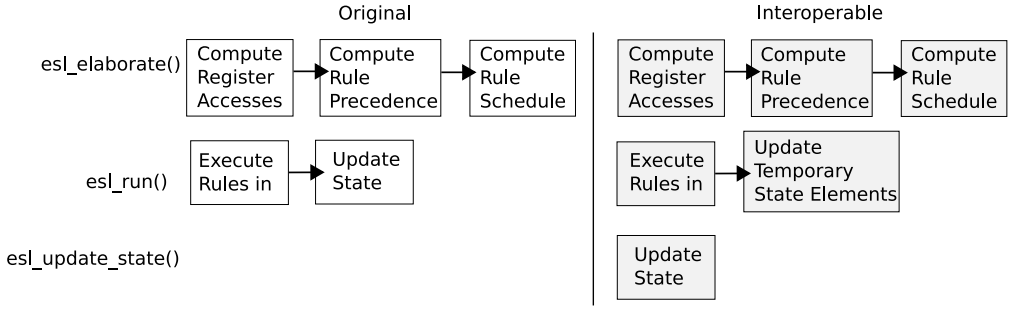


Figure 6.8: Runtime Executor Overview

We present the source code for a simple random number generator (RNG) that correctly interoperates with a SystemC driver and the wrapper code. In order to correctly simulate the RNG example, we added temporary registers within the ESL register types, *esl_reg*. Therefore, a *write* member on the register stores the parameter value into the temporary register and the *read* member returns the value in the actual register. We also added a global function, *esl_update_state*() that takes all the registers in the system and retrieves values from the temporary registers and stores it in the actual register. A schematic diagram for BS-ESL's runtime executor or kernel that implements the semantics of the rule-based MoC is shown in Figure 6.8. In the original runtime executor, the two distinct steps are the elaboration and the execution denoted by the functions *esl_elaboration*() and *esl_run*() respectively. The elaboration phase is responsible for using information about register accesses to compute *the order in* which rules may be triggered. The *esl_run*() phase executes the rules that according to the rule-based semantics are ready to execute and then updates the state elements, namely the registers. In the interoperable version, the slight modification is that the state elements being updated are temporary registers and the extra *esl_update_state*() updates the original registers with values from the temporary ones. This is only available in the interoperable version of BS-ESL's runtime executor.

6.9.2 Example

Listing 6.11: RNG Interfaces

```
1 ESL_INTERFACE( RNG_IFC ) {
```

```
2   ESL_METHOD_ACTION_INTERFACE (start, int num1, int num2 );
3   ESL_METHOD_VALUE_INTERFACE (int, get_result );
4 };
```

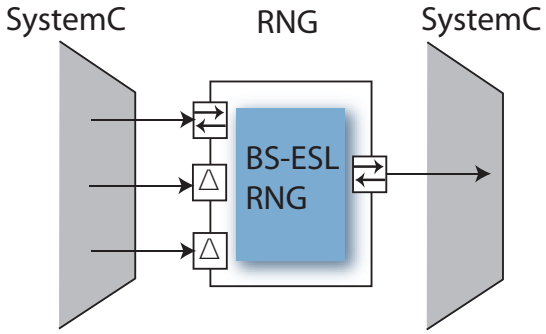


Figure 6.9: Block Diagram for Correctly Composed RNG

We continue by introducing the interface for the RNG component in Listing 6.11 that has two main members. The first initiates the RNG component via *start*, which also takes in two integer valued parameters. The second simply returns the result of the randomly generated number. The selection for both are methods except that *start* is an action method because the internal registers are updated with the values supplied by the parameters and *get_result* is a value method because it only returns the computed result.

Listing 6.12: RNG Implementation

```
1 ESL_MODULE (mkRng, RNG_IFC)
2 {
3  private:
4   esl_reg <int> p1, p2, result;
5
6  public:
7
8   ESL_RULE( rule_randomGen, true ) {
9      result.write( (( p1.read_tmp() * result.read_tmp() ) + p2.read_tmp() ) % 100 );
10  };
11
12  ESL_METHOD_ACTION (start, true, int num1, int num2) {
13    p1 = num1; p2 = num2;
14  };
15
16  ESL_METHOD_VALUE (get_result, true, int ) {
17    return result.read_tmp();
18  };
19
20  ESL_CTOR (mkRng) {
21    p1 = 0, p2 = 0; result = 79;
22    ESL_END_CTOR;
23  };
24 };
```

The RNG component implementation is shown in Listing 6.12 with implementations for the interface members as well. The three state elements are $p1$, $p2$ and *result*. The constructor initializes these register elements with initial values. This module only contains one rule *rule_randomGen* whose guard is always *true* and its action computes the result of the randomization. The remainder of the methods are the *start* and *get_result* that were explained earlier.

Listing 6.13: RNG Wrapper

```
1 SC_MODULE( rng_wrapper )
2 {
3   sc_in_clk clk;
4   sc_port< esl_delta_if > in1;
5   sc_port< esl_delta_if > in2;
6   sc_out< int>  y_out;
7  private:
8   sc_time update_time;
9   RNG_IFC *rng_inst;
10
11  public:
12   SC_CTOR( rng_wrapper) {
13     SC_THREAD( entry )  {
14       sensitive << clk.pos() <<  in1 << in2;
15     };
16     update_time = sc_time_stamp();
17     rng_inst = new mkRng ("RngModule");
18   };
19
20   void perform_update()  {
21     if ((update_time != sc_time_stamp) )     {
22       esl_update_state();
23       update_time = sc_time_stamp();
24     }
25   }
26
27   void entry() {
28     while(true) {
29       perform_update();
30       rng_inst->start( in1->read(), in2->read() );
31       esl_run(1);
32       y_out.write( rng_inst->get_result() );
33       wait();
34     }
35   };
36 };
```

Notice in Listing 6.13 and respectively in Figure 6.9 that there are two SystemC ports that are connected to Δ-channels, one clock port and an output port. The private data members are *update_time*, which is of type *sc_time* that is responsible for storing the time at which the previous call to *esl_update_state* was made and a pointer to the instance of the random number generator module. The constructor makes this wrapper process sensitive to the clock's positive edge and the Δ-channels so that any changes on those channels schedules the wrapper process to execute. It also sets the *update_time* to the current simulation time

and instantiates the RNG module. A helper member *perform_update* checks whether the simulation time at which the last call to *esl_update_state* was performed is not the same as the current simulation time, then *esl_update_state* is invoked to update the actual registers with the values in the temporary registers. Then, the *update_time* stores the current simulation time. Finally, the entry function *entry* invokes this *perform_update* member function to check whether updating all the states is required and after that the values from the input ports are transferred to the RNG module via the *start* member. This is followed by one iteration of the ESL runtime scheduler via *esl_run*(1) and later the tunneling of the result to the output. This small wrapper example ensures that the actual registers in the RNG module only update at the beginning of every simulation clock cycle. This means that any changes that occur within the clock cycles update the temporary state elements and do not affect the final result value until the simulation time is to proceed.

Table 6.2: Simulation Times for Examples

	GCD (s)		RNG (s)	
Samples	SystemC	Composed	SystemC	Composed
100000	8.0	13.2	8.62	14.7
200000	15.8	26.5	17.4	29.6
300000	24.2	39.8	26.0	43.9

6.9.3 Simulation Results

We implement the GCD and RNG examples in SystemC and in the BS-ESL co-simulation environment (labeled as composed). Our experiments showed that the correct co-simulation of composed SystemC and BS-ESL models resulted in an approximately 40% simulation performance degradation over pure SystemC simulation. This is an inevitable price for the advantage of correct by construction concurrency management in BS-ESL. The overhead in the composed models is the scheduling, data-structure construction and firing. Furthermore, the implementation in SystemC amounts to a single function call whereas in BS-ESL multiple rules are used. The overhead caused by our solution (duplicating state elements) described in this section is a fractional amount, approximately 1% of the total overhead.

6.10 Summary

This work formally presents the DE simulation semantics as implemented in any HDL including SystemC, followed by BS-ESL's rule-based semantics. Then, we argue that direct composition of the two MoCs does not provide correct simulation because of their inherently distinct MoCs. Traditional HDLs employ delta-events and delta cycles for recomputing certain functions in the designs whereas the rule-based MoC executes its rules in an ordered

manner without requiring recomputation. Using the analysis, we identify the problem with state updates during co-simulation and then present one technique that allows designers to correctly co-simulate models using the two MoCs. Our solution proposes adding a wrapper process and duplicating state elements in the BS-ESL model such that all computation is performed using the temporary state elements and the actual state elements are updated at the beginning of the next clock cycle. One of the main thrusts of this section is to show how wrapper generation should be based in formal model-based reasoning as opposed to ad-hoc ideas, as is often done for co-simulation.

Chapter 7

Model-driven Validation of SystemC Designs

SystemC [8] is increasingly being used for system level modeling and simulation during the first phases in a product's design cycle. The modeling and simulation undergoes ample validation and debugging before resulting in a functionally correct SystemC implementation model. Validation by simulation requires designers to generate a set of input sequences that exercise all the relevant and important properties of the system.

A property of the system describes a characteristic of that system. For example, a FIFO component may ignore inputs when it is full. Testing this property requires the input sequences that transition the system to that state. Thus, one needs to generate input sequences which cover trivial and nontrivial cases of a system. However, coming up with these input sequences are not always trivial. This is particularly true for large and complex designs where it is difficult to identify and describe the input sequences to reach a particular state [17, 18]. Furthermore, there may be multiple paths (different input sequences) that transition the system to the same state. Authors in [17] provide test case generation by performing static analysis on the SystemC source and then use supporting tools for generating tests. This approach is often limited by the strength of the static analysis tools and the lack of flexibility in describing the reachable state or states of interest for directed test generation. Also, static analysis requires sophisticated syntactic analysis and the construction of a semantic model, which for a language like SystemC built on C++, is difficult due to the lack of a formal semantics for it. In fact, [17, 18] does not precisely describe a semantics for SystemC. It is also difficult to diagnose the exact transition that causes a failed test case execution. For this reason, it is important to provide designers with a methodology and set of tools that ease the burden of directed test case generation and diagnosis.

SpecExplorer is a tool developed by Microsoft for model-based specification with support for conformance testing. The essence of SpecExplorer is in describing the system under investigation as a model program either in AsmL [80] or Spec# [84] and performing con-

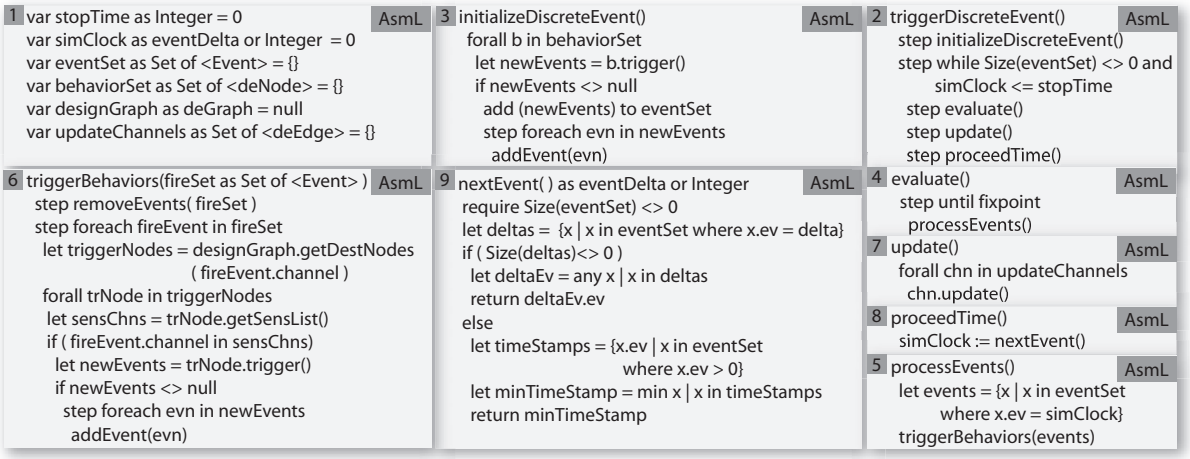


Figure 7.1: Discrete-Event Semantics Using AsmL

formance tests on an implementation model in the form of some software implementation. The model program in AsmL serves as a formal specification of the intended system because AsmL employs the Abstract State Machine (ASM) [78, 79] formalism. From here on, we use "specification" interchangeably with "model program". ASMs allow for specifying hardware or software systems at the desired level of abstraction by which the specification can focus on only the crucial aspects of the intended system. The added quality of ASM specifications being executable, aids in verifying whether the specification satisfies the requirements, whether the implementation satisfies the specification and transitively whether the implementation satisfies the requirements. This makes ASMs and SpecExplorer a suitable formalism and tool respectively for semantic modeling, simulation and as we show in this paper, for exploration and test case generation.

Previous works in [45, 46, 44] use SpecExplorer to put forward a development and verification methodology for SystemC. The difference is that they focus on the assertion based verification of SystemC designs using Property Specification Language (PSL). Their work mentions test case generation as a possibility but this important aspect of validation was largely ignored. We were not able to employ any of the work done by these authors because their tools are unavailable. Their formal discrete-event specification in AsmL, translators from PSL to AsmL, and AsmL to SystemC were not available even after several requests.

In this work, we present a model-driven method for not only specifying and developing but also validating system level designs for SystemC. We do not address the full validation of system level designs, but a small subset. In particular, we use our validation method to test corner case scenarios for which manually coming up with input sequences for testbenches may be very difficult. Figure 7.2 shows a block level schematic of the model-driven methodology. We create an abstract model from a natural language specification and this abstract model is what we call the semantic model. It is specified using AsmL (an implementation

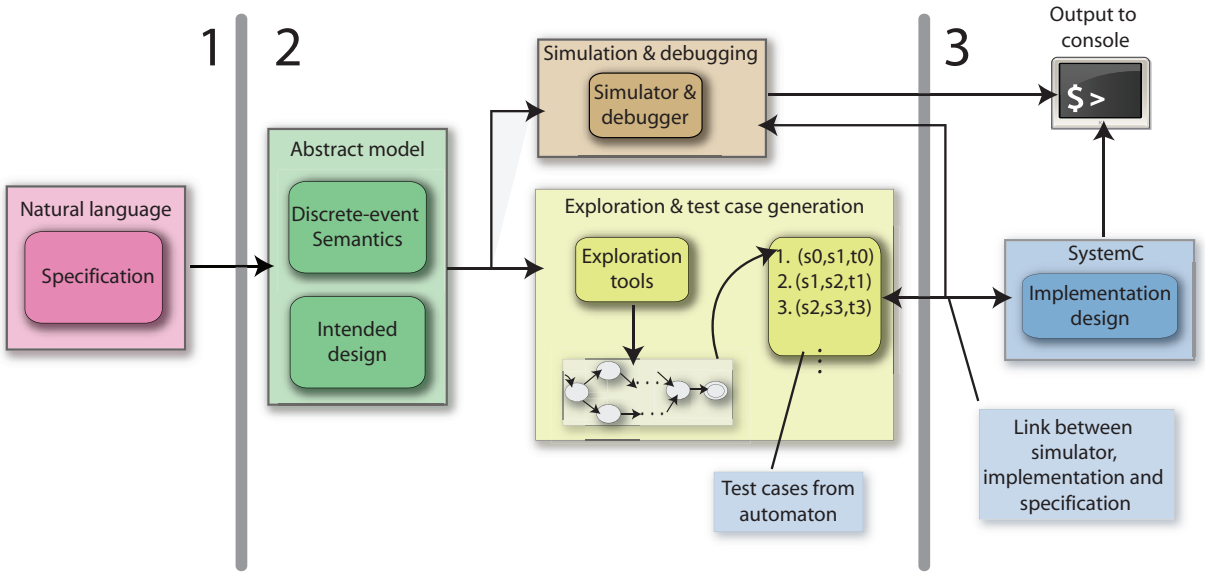


Figure 7.2: Methodology Overview

language for ASMs). Since, SystemC follows the Discrete-Event (DE) simulation semantics; we provide an AsmL specification for the DE semantics such that designers can mimic the modeling style of SystemC while using AsmL for their intended design. The specification for the intended system can then be executed with the DE specification. For testing whether an implementation model satisfies the specification, SpecExplorer provides tools for exploration and test case generation. Exploration results in generating an automaton from the specification. This automaton is then used for creating test cases. However, SpecExplorer only allows bindings to C# and Visual Basic implementation models, but for our purpose we require bindings to the implementation model in SystemC. To do this, we provide two wrappers that export functions of the SystemC library and the implementation model to libraries that are used in SpecExplorer. These functions are used in a C# interface wrapper and then bound in SpecExplorer. Figure 7.2 represents the wrappers as the link between the simulator and implementation. We use SpecExplorer's exploration and test case generation to create tests that are directed to reach certain interesting states of the intended design and at the same time executed for checking whether the implementation model conforms to the specification.

7.1 Overview of this Work

This work presents a methodology for specification, model-driven development and validation of SystemC models. This methodology is based on Microsoft SpecExplorer. In this paper:

- We provide a formal semantics for SystemC DE simulation using ASMs as semantic

foundation.

- Using Microsoft's AsmL, we provide an executable AsmL specification for SystemC DE simulation and show how abstract ASM models of intended designs can be simulated within Microsoft SpecExplorer according to our abstract DE semantics. Source code for this AsmL executable semantics for SystemC DE is available at our website [86].
- We show how the state space exploration capabilities of SpecExplorer can be effectively used for SystemC test generation for finding corner cases in the SystemC implementation.
- We show how to generate wrappers for the SystemC library and SystemC implementation model to allow SpecExplorer to drive the SystemC simulation through C#.
- We show how to do directed test generation and diagnosis for bug finding such that oversights and true errors in the implementation are found using SpecExplorer's test generation tool.

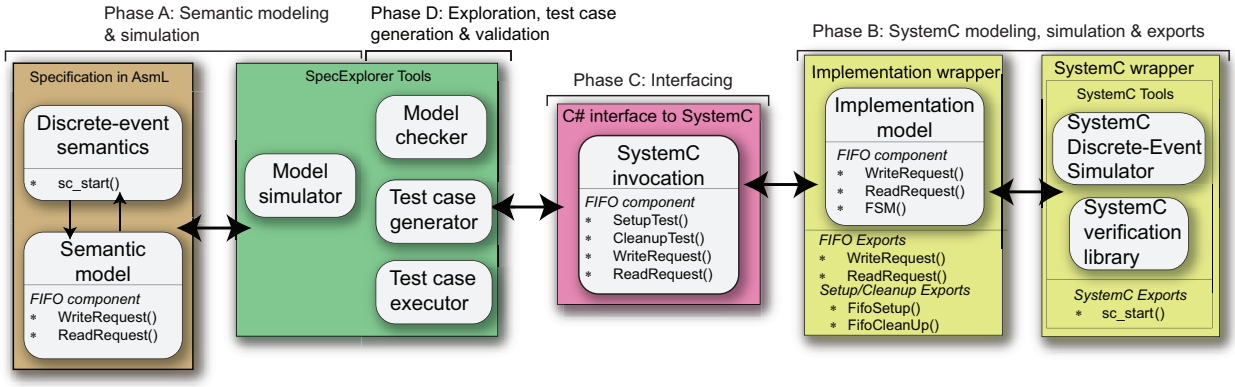


Figure 7.3: Modeling, Exploration and Validation Design Flow

7.2 Design Flow

The necessary components in employing this design flow are shown in Figure 7.3. We separate these components into four phases. These phase separations also describe the steps that a designer takes in using this methodology. Note that modular development of the system is possible and recommended, but in our description here, we assume the designer fully describes the system in each phase before proceeding to the next. This is done here to describe the methodology concisely and transparently. We also annotate Figure 7.3 with functions specific for a hardware FIFO component as our intended system (and from here on referred to it as that). We later describe this example in detail in the Section 7.3.

7.2.1 Semantic Modeling and Simulation

A typical usage mode of this methodology starts in phase `A`. In this phase, the designer creates the semantic model using constructs introduced by our Discrete-Event simulator in AsmL. For example, our semantic model of the FIFO component contains class declarations for the FIFO component, a test driver and a clock. The modeling is fashioned to be similar to SystemC's modeling style so that there can be a direct mapping from the specification to the implementation. We show two of the important functions invoked by the driver that provide stimulus for the system. They are `WriteRequest` and `ReadRequest`. These functions also specify the contract between the specification and the implementation model during test case generation. Once the semantic model is specified in AsmL, SpecExplorer's model simulator is used to validate the semantic model by simulation. One can also use some of the model checking links of SpecExplorer but we did not explore that possibility for now.

The basic validation of the semantic model is done by using testbenches. These testbenches are created by the validation engineer as basic correctness tests. Note that this is done for the implementation model as well to get a basic level of consistency between the two models. The mechanism described here is not necessary for generating these kinds of basic tests because these are the ones that evaluate basic functionalities. Once the two models pass these tests, they can be assumed to be consistent. From here, the corner case scenarios or ones for which the input sequences are hard to come up with can be generated using this directed test case generation approach.

7.2.2 SystemC Modeling, Simulation and Wrappers

Modeling

The second phase `B` is where the designer implements the FIFO component in SystemC, which we call the `Implementation model`. It also contains some of the same functions that are listed in the `Semantic model` in phase `A`. This is a result of having a clear specification of the system before moving to an implementation model. Thus, the designer has already captured the responsibilities of the members in the specification. These specification contracts translate to the implementation model.

Simulation

The implementation model is simulated with the OSCI SystemC [8] simulator and if required, any other supporting tools may be used. This is standard practice when developing systems in SystemC. As mentioned earlier, the testbenches created for the semantic model can be replicated for the implementation model to give a basic level of consistency between the model. This is important to do such that obvious discrepancies between the models do not

exist.

Wrapper Generation

SystemC Wrapper From the SystemC library, the only function that has to be exported is `sc_start()`. This function is responsible for executing the SystemC simulation. Note that this is done only once and then can be reused, because this same exported function is used for executing the simulation for different implementation models. However, we require the SystemC library to be a static library. This enables invocations of exported functions from a C# program. This does not incur any changes to the original source code but instead, we declare the `sc_main()` in an additional source code file to allow for static compilation.

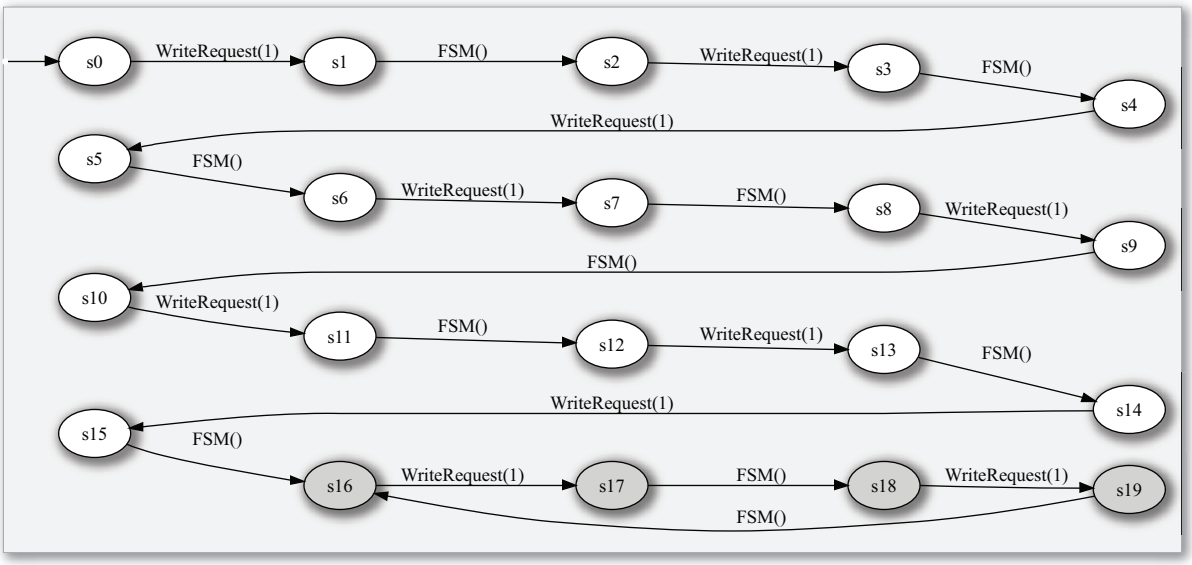
Implementation model wrapper For the implementation model the designer must be aware of the functions that stimulate the system. This is because the test cases generated in SpecExplorer will invoke these functions to make transitions in the implementation model. For the FIFO component the designer may have implemented a driver component during the modeling and simulation to control the signals for write and read requests. These inputs stimulate the FIFO component and should be the signals that are toggled during the execution of the test cases. Therefore, we first add global functions `WriteRequest` and `ReadRequest` that change the value on the respective write and read signals and then export these functions. In addition, we export two functions necessary for setting up and cleaning up the simulation denoted by `FifoSetup` and `FifoCleanUp`. The setup function creates a global instance of the FIFO component and its respective input and output signals and the clean up releases any dynamically allocated memory during the setup. Once these functions are exported, a C# program can simulate the FIFO component using the exported functions.

7.2.3 C# Interface for SpecExplorer and SystemC

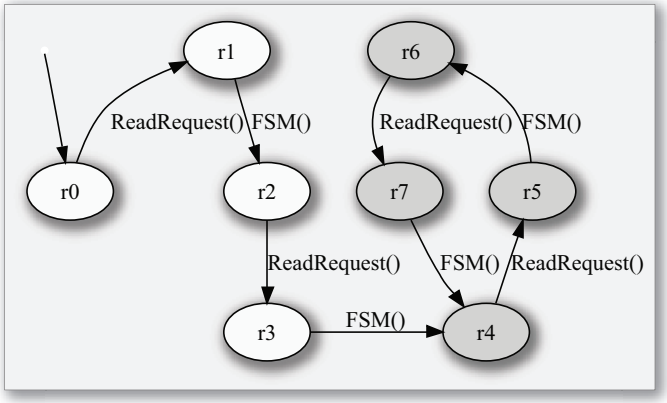
Phase `C` glues SpecExplorer with SystemC via C# such that the execution of the test cases generated by SpecExplorer symmetrically perform transitions in the implementation model. This conveniently allows the test cases to traverse the implementation model's state space as it does in the generated automaton in SpecExplorer.

The C# interface imports functions from the SystemC and the implementation model wrappers. These imported functions act as regular global functions in the C# program. We create an abstract C# class with members for the setup, clean up, requesting a write, and requesting a read. Each of these members invokes the imported functions. This allows SpecExplorer to invoke the members in the abstract C# class that in turn invoke the functions exported in the wrappers. It is necessary for the C# class members to have the exact type signatures as described in the specification. For example, `WriteRequest` takes an input ar-

gument of integer type and returns a void, so the C# class member must have this exact same type signature. If this does not conform, then the bindings from SpecExplorer are not possible. Furthermore, the C# program must be compiled as a class library to load it as a reference in SpecExplorer.



(a) Overflow



(b) Underflow

Figure 7.4: Automata Used for Validation of FIFO

7.2.4 Validation, Test Case Generation and Execution

The final phase `D` is where the test case generation and execution are done. This validation phase again requires some setup but it is necessary to have the components described in phases `A` to `C` completed before proceeding with this phase. In this phase, the designer decides what properties of the system are to be tested. Based on that decision the designer selects actions in the exploration settings and the accepting states where the tests should terminate. Then an automaton is generated. Before executing these test cases, bindings to the actions selected for exploration must be made. These actions are bound to members in the C# class library. The execution of the test cases makes transitions in the automaton generated in SpecExplorer and simultaneously stimulates the inputs in the implementation model making the respective state transitions. Inconsistencies and errors can be detected in this validation phase. SpecExplorer also shows a trace of the path taken on each test and the point at which a failed test case throws an exception. This helps the diagnosis of problems in the implementation.

7.3 Results: Validation of FIFO, FIR and GCD

We present three examples of validating designs with varying complexity. The first is the FIFO component that has been used as a running example throughout the paper. This example is discussed in detail whereas the other two examples of the greatest common divisor (GCD) and the finite impulse response (FIR) are only briefly presented.

7.3.1 FIFO

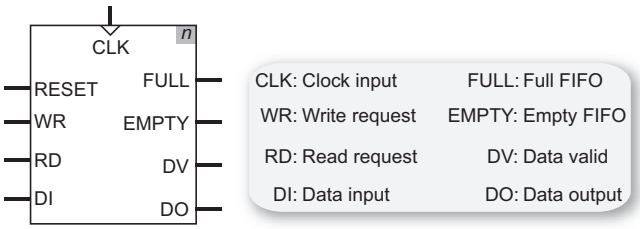


Figure 7.5: A FIFO Component

We elaborate on the FIFO component example in this section by presenting a block diagram in Figure 7.5. This is a simple FIFO component parameterized by `n` that represents the size of the FIFO. Throughout our discussion we assume `n=3`. The inputs to the FIFO are `WR`, `RD`, `DI` and `CLK` and the outputs are `FULL`, `EMPTY`, `DO` and `DV`. The `WR` requests a write onto the FIFO. If the FIFO is not full, then the data available at `DI` is stored in the FIFO. Otherwise, the data is not inserted. The `RD` input requests a read on the FIFO. This extracts the top

data element and outputs it onto `DO` while raising the signal on `DV` signifying that a data is valid. `FULL` is raised high when the FIFO reaches its maximum size `n` and `EMPTY` when the number of elements in the FIFO is zero. This FIFO component is triggered on the rising edge of the clock `CLK` and every request takes two clock cycles. The `RESET` simply clears out the FIFO and again takes two clock cycles.

```
class requestGenerator extends deNode                    AsmL
  requestGenerator( name as String )
    mybase(name)
  override trigger( ) as  Set of Event?
    step
      WriteRequest(1)
    return null
  ReadRequest()
★ reqmode = READWRITE
★ step
★  reqmode := PROCESS
  step
    RD.value := true
  WriteRequest(data as Integer)
★ require reqmode = READWRITE
★ step
★  reqmode := PROCESS
  step
    WR.write( true )
    DI.write( data  )
```

```
class FIFOBlock extends deNode                           AsmL
  FIFOBlock( name as String )
    mybase( name )
  Full()
    if ( is_full() = true)
      FULL.write( true )
    else
      FULL.write(false)
  Empty()
    if ( is_empty() = true )
      EMPTY.write(true)
    else
      EMPTY.write(false)
  Reset()
    if (is_empty() = false)
      clear()
  override trigger( ) as  Set of Event?
    step
      FSM()
    return null
```

```
FSM()                                                    AsmL
★ require (reqmode = PROCESS )
★ reqmode := READWRITE
  if (mode = INIT)
    mode := REQUESTS
  if (mode = REQUESTS)
    step
      Full() // Update FULL status
      Empty() // Update EMPTY status
    step
      if ( WR.read() = true and FULL.read() = false )
        push(DI.read()) // Throws excep. overflow
        WR.write( false )
      else
        if ( RD.read() = true and EMPTY.read() = false )
          DO := pop() // Throws excep. overflow
          DV.write( true )
          RD.write( false )
    mode := WAIT
  if ( mode = WAIT)
    DV.write( false )
    mode := REQUESTS
```

Figure 7.6: FIFO Specification in AsmL

AsmL Specification and SystemC Implementation

The AsmL specification for the FIFO component employs the DE semantics that is also described in AsmL. This is shown in Figure 7.6. The two basic components are the `requestGenerator` and the `FIFOBlock`. As the names suggest, the former serves as a driver for the latter. The member name of the respective classes is overlayed with a box in Figure 7.6. In the class declaration of `requestGenerator`, we start with the constructor that simply invokes a base class constructor of the inherited class `deNode`, which is a class available from the DE specification. The `trigger` member is in terms of SystemC terminology, the entry function of the component, or in other words, the member that is invoked by the simulator. The remaining two members, `ReadRequest` and `WriteRequest` perform requests on the FIFO component. For the `FIFOBlock`, we again present the constructor followed by three members that perform the `Full`, `Empty` and `Reset` operations. The entry function of the `FIFOBlock` component only invokes the internal member `FSM`.

This `FSM` member presents the crux of the FIFO component's behavior as an FSM. The FSM has three states `INIT`, `REQUESTS` and `WAIT`. The initial state of the FSM is `INIT` after which it enters the `REQUESTS` state. Upon receiving a request of either read or write, the `FULL` and

`EMPTY` states are updated via the `Full` and `Empty` functions. If a write is requested, then the respective states and flags are updated and the same is done if a read was requested. The write request is given priority whenever both requests are simultaneously made. After the requests are serviced, the state of the FSM is changed to `WAIT`, which is when some of the state variable values are reset and the state is returned back to accepting requests at the next step. Note that the `WAIT` state in the SystemC implementation model could use SystemC's `wait` if a SystemC SC_THREAD process type is used. In our implementation model we use SC_METHOD SystemC processes and hence the need for an internal `WAIT` state. Also note that we do not present the AsmL code that instantiates and connects the two components since that is relatively straightforward. The lines marked with a $\star$ (star) are either altered or added only during exploration to prune the state space and only result in the states and transition that are desired for a particular validation scenario. We explain this in more detail in the exploration section

Once the semantic model is described, it is important to validate the semantic model via simulation. This requires coming up with testbenches that test the basic functionality of the FIFO. In doing this, the validation engineer can test that the expected behavior of the system is modeled. For example, a request for a write actually does do a write and similarly with the read. However, note that the problem of coming up with input sequences for the difficult properties still remain. We repeat the testbench simulation approach for the implementation model as well. Again, the reason for this is to pass the basic tests to ensure that the easy to test properties hold. This provides a limited level of consistency between the two models. From here, we use the exploration approach for the more difficult test scenarios.

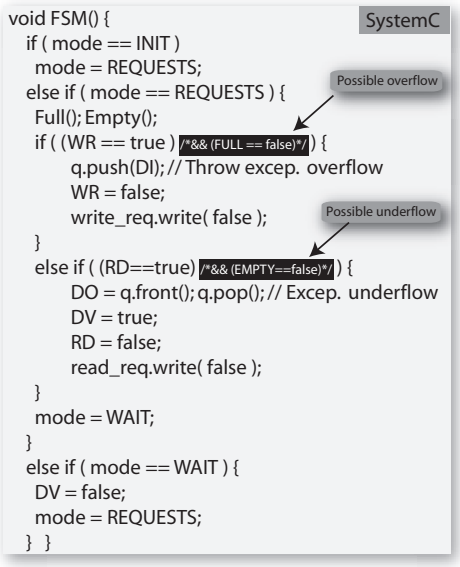


Figure 7.7: SystemC FIFO FSM Code Snippet

Exploration and Test Cases for Specific Properties

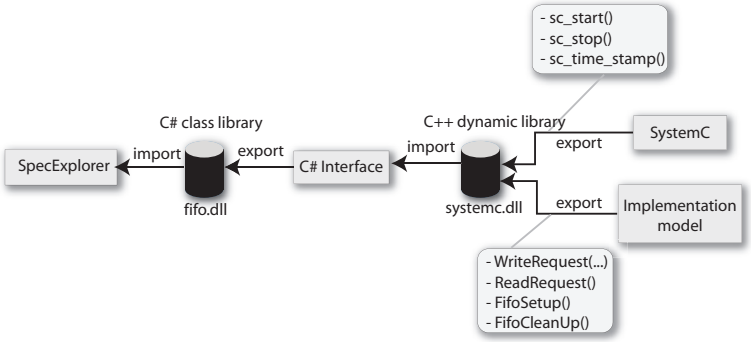
Notice in Figure 7.7 the SystemC code that describes the FIFO's FSM implementation. We have intentionally commented out code fragments in this figure such that it is possible to cause an overflow and an underflow situation. This depicts possible modeling errors and bugs in the implementation. Now, we present test case generation for validating whether these two basic but essential properties of the FIFO hold. The overflow situation occurs when a write is requested and when the FIFO is full. An underflow occurs when a read is requested and the FIFO is empty.

In order to generate test cases for either of the properties we need to explore the specification. Exploration using SpecExplorer requires the use of several abstraction techniques since the semantic model may suffer from state space explosion. During this procedure, it is important to understand how the automaton is generated from ASM specifications. We forward the readers to [110] for details on the algorithms for generating automata from ASMs, but we point out that the automaton generated is based on the specified actions and states of interest. It is not generated via the execution of the entire semantic model, but simply the actions specified.

We present details on some of the necessary techniques used in exploring the FIFO example. The techniques presented here are used for other properties and semantic models. However, we only present the detailed discussion only for the overflow property. We start by assuming that the $\star$ statements in Figure 7.6 are not present in the semantic model. So, for generating test cases for the overflow property, we perform `n+1` number of successive writes without any read requests.

For successful write requests we make `WriteRequest` and `FSM` controllable actions in the exploration configurations. This is because `WriteRequest` is responsible for issuing the request and `FSM` is the member that actually updates the internal FIFO. Since we want to essentially test for write requests until the `FULL` signal is `true`, our accepting states are those in which `FULL.read() = true`. Therefore, in the exploration settings we set the accepting states condition with the expression for showing the FIFO is full. Notice that the `WriteRequest` takes in an argument of type integer and by default SpecExplorer assigns a set of possible integers for the integer type. This can be changed to whatever is desired. For simplicity, we change this default value of type integer to just `1` to avoid creating an automaton with all possible paths with all integer inputs.

This above exploration configuration results in an invocation of each action from every state. This may be the correct automaton desired by the validation engineer, but suppose that for our investigation, we want to explicitly generate the automaton that first invokes `WriteRequest` followed by `FSM` where each invocation of the action results in a new state. This is where we bring in the statements that are shown next to a $\star$ in Figure 7.6. To generate the desired automaton that performs `n+1` write requests, we overlay the `WriteRequest` and `FSM` with a simple but additional state machine using `reqmode` that updates the `reqmode` state



(a) Wrappers and Libraries

Example of exporting SystemC function

```
extern "C" _declspec(dllexport)   C++
        int hdp_sc_start(int duration);
int hdp_sc_start(int duration) {
        return sc_start(duration);
}
```

Example of importing C# abstract member

```
[DllImport(@"C:\validationASM\systemc.dll",   C#
        ExactSpelling = false)]
public static extern
        int hdp_sc_start(int duration);

 public abstract class SystemCFifoBlock   {
    public static void FSM()     {
      if (hdp_sc_start(1) != 0)     {
        // handle error
      }
    }
 };
```

(b) Export Code Snippets

Figure 7.8: Wrapper Construction and Export Code Snippets

element with every invocation of the actions. The $\star$ lines are added to support this overlaying state machine. However, notice that we use `require` that basically is an assertion that unless the expression evaluates to true, the member cannot be invoked. The loop-transitions are removed by introducing the `require` statement because the automaton generation algorithm only explores the respective state when the transition enabling the `require` evaluates to true. This further refines the automaton yielding the corresponding automaton with these additional changes in Figure 7.4(a). The accepting states are shaded and every transition performs a write request with a value `1`. It is possible to vary the values as well but the concept is easily understood by simplifying the automaton with only one input value. The additions to the semantic model produce the desired automaton. Also remember that due to the internal state machine controlled by `mode`, it takes two invocation of `FSM` for a successful write on the FIFO.

Executing test cases generated from this automaton raises an exception when the fourth

successful write request is made. SpecExplorer's diagnosis indicates that the error occurred when taking the transition from state s_{15} to s_{16}. This suggests that there is a discrepancy between the semantic and the implementation models at the particular state when the FIFO's state is updated to full and there be a write request. SpecExplorer provides an interface for viewing and following the transitions taken in the test cases before the exception was thrown. This is important because it makes it easier for a designer to locate the transition at which a possible error exists. Furthermore, a backtrace to the initial state shows the diagnosis or the input sequence that leads to this erroneous state. This is advantageous for the designer because the error can also be duplicated. Evidently, the code fragment marked as a possible overflow situation in Figure 7.7 causes this erroneous behavior. Removing this yields in successful execution of the test cases.

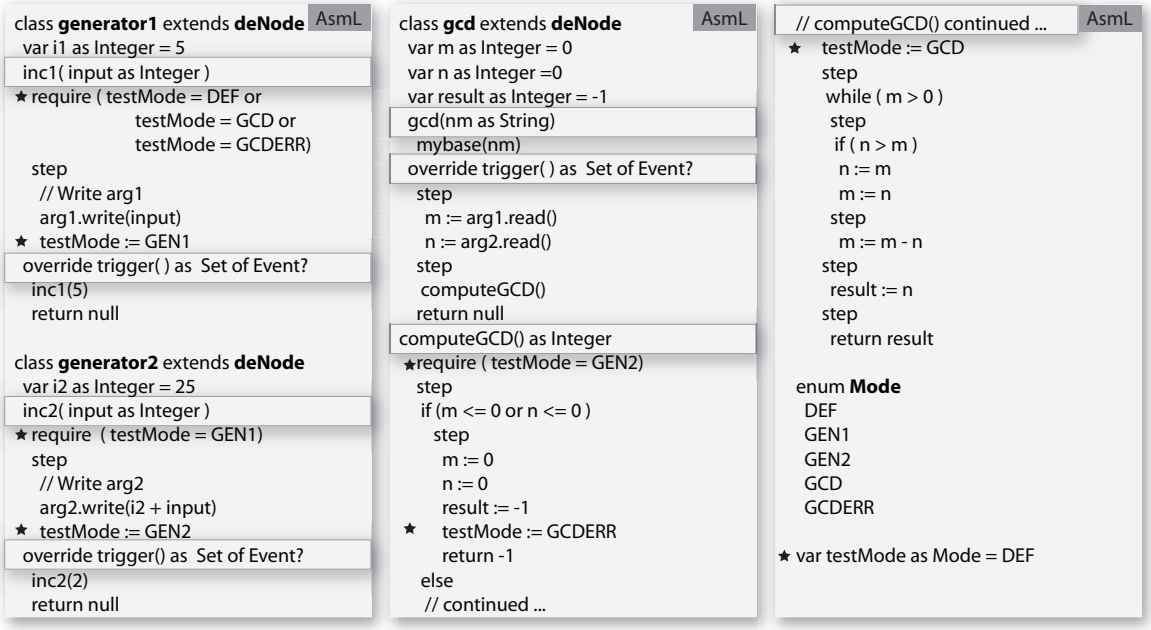


Figure 7.9: GCD AsmL Specification

Generating test cases for validating the underflow property is done using a similar approach. The required steps in checking for underflow require the following: 1) adding `ReadRequest` and `FSM` as controllable actions in the exploration settings and 2) making the accepting state to be when the `EMPTY` state variable evaluates to true. The resulting automaton is shown in Figure 7.4(b).

Executing the test cases for this automaton again shows that there is a violation in the implementation model. This occurs at the first successful `ReadRequest` on the transition from state r_3 to state r_4. This is expected because the `EMPTY` state is updated to realize that the FIFO is empty and then the first read request is serviced, but the FIFO has no elements stored in it. Since our implementation model shown in Figure 7.7 has the error marked as

a possible underflow, the implementation throws an exception at this first successful read request. Once again, by uncommenting the erroneous fragments, the test cases all succeed.

Wrapper Generation

An integral part to the design flow is the construction of wrappers between SpecExplorer and SystemC. Figure 7.8(a) shows the process of creating the wrappers and its associated libraries. We describe the flow for generating wrappers and exporting members from the implementation model such that they can be used in the semantic model.

From the SystemC distribution it is necessary to export the `sc_start`, `sc_stop` and `sc_time_stamp` members where the first drives the simulation, the second stops the simulation and the third simply serves for debugging purposes. From the implementation model, the necessary members that drive the simulation in that model must be exported, which for the FIFO example are listed. These two sets of exported members are combined into a dynamic C++ library and a snippet of the C++ code used for exporting and importing it into C# is shown in Figure 7.8(b). The C# wrapper interface imports the `systemc.dll` library for the specific exported members. These are then used in the interface definition in C#. For example in Figure 7.8(b), we show the definition of the `FSM` member in C#. Note that the class that this member is contained in is an abstract class and that the types of the members must match those in the semantic model. This C# interface is then compiled as a class library that is called a reference in SpecExplorer. This reference is loaded into SpecExplorer so that test action bindings can be made.

7.3.2 GCD

The GCD example consists of two input generators and a computation unit. The input generators provide input for the GCD computation and keep changing the inputs. In our case the input generators simply write an integer value. The GCD computation unit follows Euclid's algorithm. Figure 7.9 describes the AsmL specification that uses the DE simulation semantics also implemented in AsmL. The components described by `generator1` and `generator2` are simply writing to a channel `arg1` and `arg2` respectively. The real computation happens in the `computeGCD` member of the `gcd` component. We have a corresponding implementation model for the GCD that is not shown here, but it is available at our website [86].

Exploration and Test Case Generation

The exploration techniques used for generating the automaton for the GCD example are similar to that explained in the FIFO example. Therefore, without much discussion on the exploration details, we list the two properties that we are interested in validating. They are:

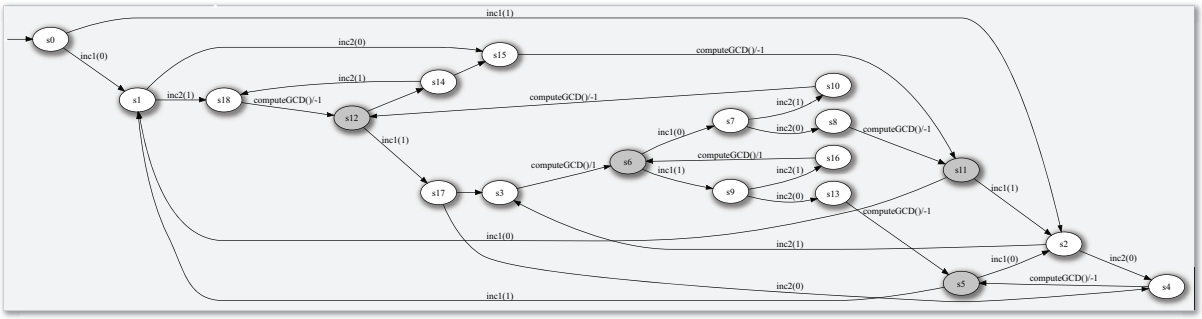


Figure 7.10: Automaton for Validating GCD with 0 or 1 as Inputs

- the intended operation of the GCD computation component when encountering invalid input data and
- simply validating when the input sequence consist of either `0` and `1`.

To facilitate the exploration for validating the above two properties, we overlay a state machine described by the enumerated type `Mode` standing for request mode and the variable `testMode`. This overlaying procedure is once again similar to that of the FIFO example. Note that in the `computeGCD` member, the result is assigned a `-1` value when there is an erroneous computation request. For this to occur, either of the inputs must be zero or any non-negative integer. So, to validate the first property we do the following: 1) add `inc1`, `inc2` and `computeGCD` as controllable actions, 2) set `testMode = GCDERR` as the accepting states condition and 3) change the default integer values for the parameter for `inc1` and `inc2` to `-1`. The resulting automaton is shown in Figure 7.11.

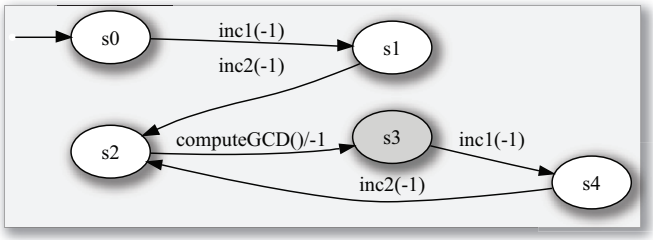


Figure 7.11: Automaton for Validating Invalid Inputs

The transition from state s_2 to s_3 is `computeGCD()/-1`, which is in the form of `action/result`. This means that when the `computeGCD` action was invoked, the result of that member was `-1` and this should match with the implementation model.

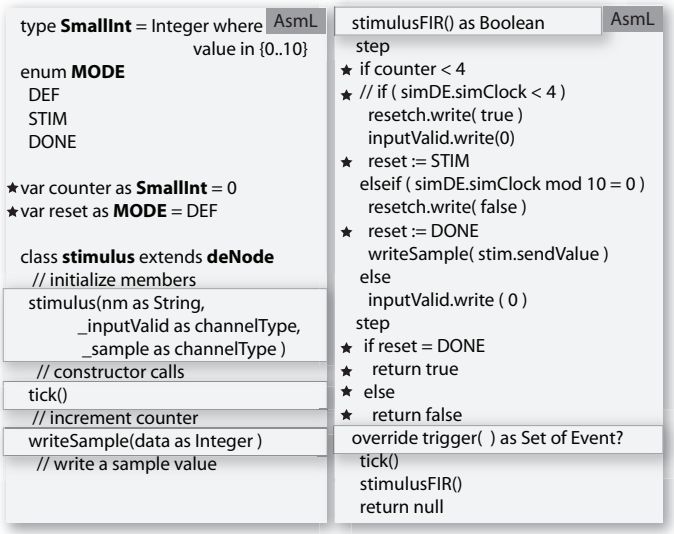
The validation of the second property requires slight alteration to the exploration configuration. The default values for the integers that are used as parameters to `inc1` and `inc2` and the accepting states condition are altered. We change the default values to allow for `0`

and `1` and set the accepting states expression to `testMode = GCD or testMode = GCDERR`. The accepting states are all states where there is a computation that results either in an erroneous or correct computation. The automaton from this exploration configuration is relatively large in size and shown in Figure 7.10.

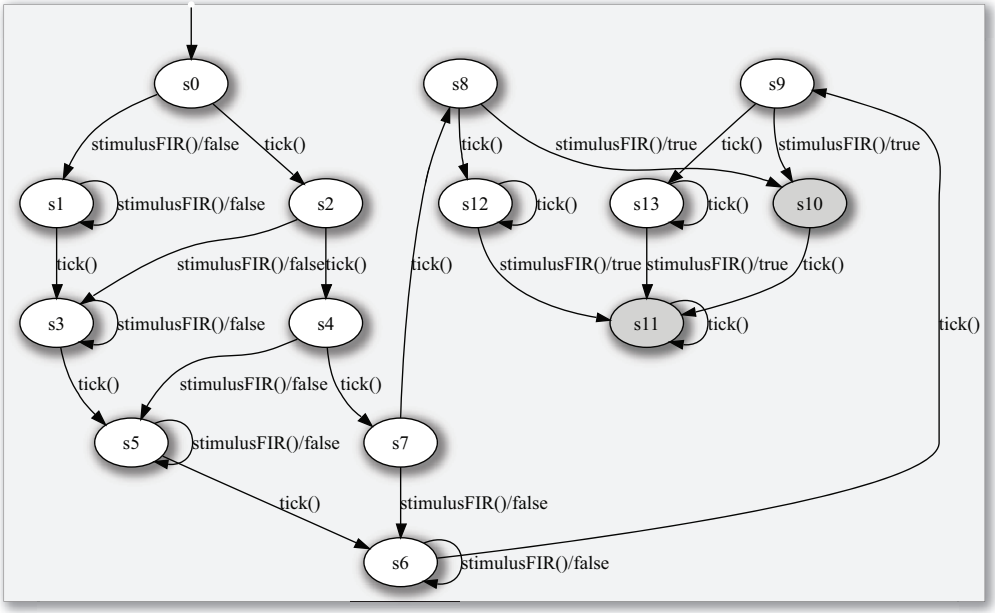
7.3.3 FIR

The finite impulse response semantic model is based on the FIR example from the SystemC distribution. The FIR has a stimulus, computation and display component. An important property that must hold in order for the FIR to correctly operate is to initiate the reset for a specified number of cycles. The entire FIR requires four clock cycles for the reset to propagate. This is an interesting example because the validation requires exploration techniques that are different from the FIFO and the GCD example. We present snippets of the AsmL specification for the stimulus component used in validating this property in Figure 7.12(a).

The two members of importance are the `tick` and `stimulusFIR` that increment the cycle counter `counter` and compute whether the reset has completed `reset` respectively. These are the members that are added as actions in the exploration configuration. So, our exploration settings have (1) `tick` and `stimulusFIR` as controllable actions and (2) `counter` and `reset` as representative examples of a state grouping. However, note that we introduce a new constrained type `SmallInt` for the `counter` variable. This constrained type is a subset of the integer type with the possible values it can take from `0` to `10`. This cycle `counter` is incremented in `tick` and used for checking the cycle count in `stimulusFIR`. In our semantic model we had used the DE simulator's internal clock for comparing the cycle count, but we alter this for exploration purposes. This is because `simClock`'s state is updated within a member of the simulator, mainly `proceedTime`. Therefore, for the automaton to reflect the change in state of `simClock` additional unnecessary simulation members have to be added and to avoid this, we simply replace it with the `counter` variable and add `tick` to increment the cycle counter.



(a) FIR Stimulus Component AsmL Code Snippets



(b) Automaton for Validating Reset for Four Cycles

Figure 7.12: FIR Example

The automaton generated from the above exploration configuration is shown in Figure 7.12(b). Executing the test cases generated for this automaton show that the reset is indeed held high for four cycles thus showing conformance between the semantic and implementation models. The full source for the semantic model, implementation model and the wrappers are available at [86] along with the other examples.

7.4 Our Experience

The first phase of semantic modeling is intuitive and simple. This is because of our additional DE simulation semantics in ASM that makes it simple for designers already familiar with SystemC or any other DE simulation environment to describe their semantic models. The modeling paradigm and extensibility of the DE semantics allows for user-defined channels and modules. The semantics also take into account the non-determinism mentioned in the SystemC IEEE 1666 standard. However, we do not model all constructs available in SystemC. For example the `wait` statements are not supported in our version of the AsmL discrete-event semantics. The authors of [47] describe how they incorporate `wait` semantics by instantiating program counters for each thread and saving the counter when switching out. However, any thread-based behavior can be easily modeled as a method-based model [111] and thus refrain from extending our semantics for the sake of clarity. The simulation and debugging of the semantic models is again quite similar to SystemC simulation and debugging with the addition that SpecExplorer allows for step-by-step debugging.

Our experience in creating the implementation models was positive because of the similarity between the semantic and implementation models. In fact, we experienced an easy translation from the semantics to the implementation model. The aspect of simulation and debugging is similar to that at the semantic level.

The challenging aspect of this methodology was the exploration of the semantic model. This requires a thorough understanding of how automata are generated from AsmL specifications and how to steer it using exploration techniques to validate the desired property. Managing the state space is difficult in complex designs and understanding the exploration techniques in SpecExplorer is important to use the state space pruning techniques. Our experience suggests that after exploring on a few examples the techniques and capabilities of SpecExplorer can be understood clearly thus making the exploration much simpler for a designer or validator.

7.5 Evaluation of this Approach

This section describes the advantages, disadvantages and limitations of the proposed model-driven validation approach for SystemC designs. We do not present a discussion on the benefits and limitations over coverage-driven test generation or random test generation.

This is because the purpose of this work is for validating specific corner case scenarios. This test generation method should augment existing test generation methods for the purpose of directed test case generation.

7.5.1 Benefits of this Approach

Using AsmL for semantic modeling brings with it the advantage that it is based on the Abstract State Machines formalism. The formal nature of ASMs means that the semantic model described in AsmL is a formal specification of the design. So, any critical properties of the intended design can be manually proved by presenting proofs on the AsmL specification. ASMs also define operational semantics that describe the execution of ASMs. SpecExplorer's implementation of these operational semantics makes any semantic model described in AsmL executable. This gives engineers a formal modeling and simulation framework based on AsmL, without having to be an expert in the use of formal methods. More importantly, it is a formal specification similar to the finite state machine formal model with additional support for fine grained parallelism. Assuming that engineers are familiar with finite state machines, a formalism based on state machines should be easy for engineers to adopt.

The advantage of providing the AsmL specification of the Discrete-Event simulation semantics is that engineers can write their intended design in a fashion that is similar to SystemC. This involves describing behaviors in terms of processes and communication through channels. Moreover, this AsmL semantic model can then be simulated using the Discrete-Event semantics as it would be in SystemC. This is ideal for engineers familiar with the simulation semantics of SystemC.

The SpecExplorer tool makes it easy to specify the transitions of interest denoted by functions that change values of state variables, the states of interest and the accepting states. These are used in generating an automaton from the exploration of the design. There is a graphical user interface (GUI) for visualizing the generated automaton. This allows an engineer to step through the automaton and look at the values of the state elements of interest. However, for large automata this may be difficult. So, there is facility to export the automaton to an XML file for other visualization tools to use or even feeding it as input to model checkers.

Generating tests from this automaton is a simple click-button approach in SpecExplorer. The tool provides a variety of methods for generating tests from the automaton, such as random walk sequences or all possible sequences from initial to final states. These test sequences can also be exported to an XML file. The execution of tests simply goes through the list of test sequences traversing the automaton. Assuming that the wrappers are available, it is easy to connect SpecExplorer to the imported C# library via its GUI. During the execution of the tests in the semantic model, the functions that are bound to the functions imported from the C# library are also executed. This simultaneously executes the two models with the same input sequence. This is one of the important advantages of this approach over traditional approaches, where the engineer has to come up with an input sequence to reach

a set of desired states in the implementation model.

Note that a significant advantage of using SpecExplorer for test generation is that the engineer has to represent the test scenario in the form of an automaton. This requires exploration of the semantic model. The exploration consists of indicating the states that are relevant (in terms of the ones that will be updated in testing this scenario), the functions that transition states and the Boolean expression dictating the accepting states. The input sequences for the tests are generated by traversing through the automaton. In situations where the state space becomes large, it is possible to overlay the semantic model with a much more abstract state machine. This state machine can then be used as a representative sample of the states of the design for generating an automaton. The input sequences are determined by the arguments of the functions that transition the states. So for example, `WriteRequest(req as Integer)` is the prototype of a function for the FIFO example that issues a write reqeust with value `req`. Suppose that the FIFO could only take values encoded by a subset of the Natural numbers {0,1,2}. Then, we can force SpecExplorer to use the default values of Integer as the subset just described. This will create three different transitions with the values 0, 1 and 2, from every state that can invoke a `WriteRequest`. Traversing all paths from the initial to the accepting states gives all possible input sequences to the accepting states. This provides validation engineers with a systematic method to describe the test scenario in the form of an automaton and let SpecExplorer generate the tests by path traversal of the automaton.

There is support for diagnosis in SpecExplorer. This is used when a discrepancy between the semantic and implementation model is discovered. This results in an exception that is caught by SpecExplorer. The GUI and the conformance logs (list of sequences of a test to be executed) can be used to step through the automaton to distinguish which transition caused the exception. This is a good method for debugging and finding the exact discrepancy between the two models.

7.5.2 Drawbacks of this Approach

The foremost disadvantage in the model-driven approach is that both the semantic model and the implementation model have to be constructed by the engineer. An engineer may spend more time in creating the additional semantic model than simply creating the implementation model and manually creating tests for that model. This is particularly true for designs that are not complex. Ideally, designs that have complex state transitions and a large state space are good candidates for the model-driven test generation approach. We acknowledge that currently, this is an additional cost in the design process.

A problem that follows is that there may be inconsistencies between the semantic and implementation model because they are created separately. However, the method flow suggests that the modeling and simulation takes place at both the semantic and implementation model. This means that these basic tests can be validated via testbenches on both the ab-

stractions. By simulation we know that the two models are consistent. However, there may still remain properties that are hard to validate because the sequence of inputs to test these properties may be difficult to come up with. These corner case or safety-critical properties can use the directed test case generation approach via exploration. This provides a method for the engineer to discover this input sequence.

As we mentioned, we reserve this directed test case generation approach for specific corner case properties for which the input sequences are difficult to discover. So, we do not solve the entire validation problem of ensuring that every propery extracted from the natural language specification is tested using this approach. Rather, many properties are easy to test using testbenches. So, our approach tackles a very specific and partial solution to the entire validation aspect of an ESL methodology.

Another disadvantage is that in most model-driven design flows without support for automatic generation of implementation model, the abstract representation is often times discarded. This makes the implementation model the reference model, where any changes or updates to the specificiation are properly reflected. This leaves the semantic model disconnected from the implementation model; effectively making this test generation method unusable. This rises the importance of being able to generate the implementation model from the semantic model. In fact, the authors of [45, 44] have hinted, that it is possible to create compilers that translate AsmL descriptions to the implementation model. This requires giving formal semantics to SystemC and then mapping the ASM semantics of the AsmL specification to SystemC semantics. We feel that this can be done to reduce the cost of having to create both the models and allowing design flows to discard the semantic model. Assuming that this translation from AsmL is done, the engineer only has to create the semantic model from which an implementation model is automatically created. Any changes to the design are reflected in the semantic model from which the implementation can be automatically updated via the translation tools.

The wrapper generation is currently an arduous task because, for every intended design, the wrappers have to be manually generated. There are also two levels of wrappers that are needed; one exporting the implementation model's functions and the other making the C# class library. These two levels are necessary because SpecExplorer can only interface with C# or Visual Basic, and SystemC is designed using unmanaged C++. So, there is a lot of unnecessary overhead in creating the wrappers to connect the functions of interest in the semantic model to the implementation. However, it is possible to create scripts that will automatically generate the source code needed to export the members from the implementation model and automatically create the C# class library. We have not done this here, but Perl scripting can be effectively used to perform this.

This brings us to a significant drawback of this approach, which is that the entire validation approach is based around the SpecExplorer tool. This tool is only for Microsoft operating systems and even though SystemC is available for both Unix-flavored and Microsoft operating systems, it is commonly used in Unix environments in industry. In our opinion, there are

two possible ways of resolving this issue. The first is that we use a distributed software system approach by using C# to interface through an implementation of CORBA such as Borland VisiBroker [112]. At the same time, SystemC can be used to interface with a CORBA implementation in Unix systems. Thus, allowing the distributed systems of SpecExplorer and SystemC to communicate through an object request broker. This resembles the approach we discuss for the dynamic integration of multiple tools. A second alternative involves replicating the features of SpecExplorer in a Unix environment. There are several open-source implementations of the ASM semantics such as XASM [113] and CoreASM [114] that may be extended with similar facilities provided by SpecExplorer.

One other limitation of this test generation method has to do with the GUI for visualizing the automaton in SpecExplorer. The number of states and transitions quickly increases, making it difficult to maneuver using the GUI. SpecExplorer has certain exploration techniques that can be used to better represent the state space such as state groupings. These help in grouping states of interest together so that the resulting automaton is manageable. This above mentioned issue is coupled with the problem of state space explosion. For this, it is necessary that the engineer has enough experience in the exploration techniques provided in SpecExplorer. This requires practice and some trial and error, which can be an added learning cost. For example, a semantic model that either takes a large amount of time to generate the automaton or one that cannot generate an automaton in the specified number of bounds (for both states and transitions) must be abstracted further. By this we mean that the engineer must introduce finite state machines that overlay the semantic model and make the state elements of this finite state machine the states of interest. The downside is that the semantic model has changed now, which means that it must be simulated again to check for correctness. This limitation of requiring knowledge about the exploration techniques needed to generate the automaton is unavoidable. However, the exploration approach in SpecExplorer does provide the engineer with a method for directing the automaton to test the desired final states as opposed to the engineer having to know the implementation of it and steering the inputs without any additional tools.

Finally, we have only shown our validation method used for SystemC, which follows the Discrete-Event simulation semantics and not for the other heterogeneous models of computation (MoC)s. For us to enable this, we need to provide formal semantics for each of the MoCs in AsmL and allow the semantic model to represent these MoCs. Then it would be possible to describe the semantic model using these MoC descriptions in AsmL and via the extensions for SystemC in the implementation model. Once these are available, we can model and simulate at both the semantic and implementation level (using the extensions for SystemC). The validation method that follows is the same as proposed when using the Discrete-Event MoC. Except that there are multiple MoCs allowed in the framework. We find that this is an implementation exercise, and feel that the proof of concept is sufficient to show that as future work it could be extended to allow for heterogeneous MoCs.

7.6 Summary

We present a model-driven methodology for validating systems modeled in SystemC. Our approach uses the formal semantic foundation of ASMs via AsmL for semantic modeling and simulation. This formal specification captures the requirements of the intended system at a high abstraction level and defines a contract with the implementation model. The formal specification in AsmL helps in serving any necessary proof obligations required for the designs as well. The designer can then follow this specification when creating the implementation model and since ASMs describe state machines, the mapping to SystemC implementation is intuitive and natural. After the wrapper generations, SpecExplorer's test case generation can be directed to generate test cases for reaching interesting states in the system. A diagnosis is also provided on these test case executions. SpecExplorer has previously been used for proposing verification methodologies for SystemC designs, but not for test case generation and execution. This is an important addition to the validation of SystemC designs. We also show examples of directing the test case generation for the hardware FIFO component, GCD and the FIR. The FIFO example discusses in details the exploration techniques and wrappers necessary in employing this methodology. Even though we present our methodology as a model-driven validation approach from the semantic model to the implementation model, it is possible to write semantic models from existing designs and then the semantic model can be used for test case generation purposes. This is the case for the FIR example that we present.

Our overall experience in using this methodology has been positive in evaluating whether the semantic and implementation models conform to each other. The main difficulty as discussed was in knowing how to use SpecExplorer and its methods for state space pruning and exploration. This is especially true for large and complex designs. However, this methodology scales effectively due to the ease in raising the abstraction level using ASMs. This makes it much easier to focus on only the essential aspects of the system. Furthermore, after familiarizing with the techniques of state space pruning and exploration, the task of directing the test case generation is routine. Our semantic and implementation models and wrappers will be available on our website.

Chapter 8

Service-orientation for Dynamic Integration of Multiple Tools

The rising complexity of embedded system design and the increasing engineering efforts for their realization has resulted in a widening of the productivity gap. Efforts towards mitigating the productivity gap have raised the importance of System Level Design (SLD)s languages and frameworks. In recent years, we have seen SLD languages such as SystemC, SpecC, SystemVerilog [8, 115, 12] in efforts to raise the level of abstraction in hardware design. These SLDs assist designers in modeling, simulation and validation of complex designs. However, the overbearing complexity and heterogeneity of designs make it difficult for embedded system designers to meet the time-to-market with just the provided SLDs. Hence the proliferation of numerous commercial tools supporting SLD languages with features for improved model building experience. This shows that designers need improved techniques, methodologies and tools for better debugging, visualization, validation and verification in order to improve productivity.

Most of the present SLD languages and frameworks (SLDL) are primarily equipped with a modeling and simulation environment and lack facilities to ease model building, visualization, execution analysis, automated test generation, improved debugging, etc., which are becoming increasingly important to enhance the overall design process. As evidenced by the release of commercial tools such as Debussy [21], ConvergenSC [22], VCS [116], Incisive [24] to mention a few, SLDL themselves require additional supporting tools. However, with the multitude of commercial tools, a designer must select the set of tools that best fits his/her needs and many times the existing tools do not suffice. This raises an issue of extendibility, where designers have no way to extend the commercial tools or solutions to suit their own specific needs. Even if the SLDL can be subject to alterations, the abundance of open-source tools are subdued by the issue of efficient deployment mechanisms and reusability. In addition to the ability of designers to extend a framework, they should also be able to cleanly interface and distribute their solution for integration with other SLDLs.

Some industrial outcomes to addressing these inadequacies in SLDLs are simulation-based dynamic validation frameworks such as ConvergenSC [22], VCS [116] and Cadence Incisive [24]. However each of these tackle various aspects of the design process. There are some overlapping features and some distinct desirable features, and at this moment it is not possible to unify the capabilities into one framework. Furthermore, the inherent problem of extendibility, disallows users to consider altering or integrating these different frameworks.

In efforts to address these issues with industrial simulation-based dynamic validation frameworks and the lack of features for SLDLs, we propose a simulation based dynamic validation framework called CARH[1] as a solution for SystemC. CARH is a service oriented verification and validation framework using middleware technology. In particular we employ TAO [63] that is real-time (R/T) CORBA Object Request Broker (ORB) [59] and ACE [61], to allow for a language independent pluggable interface with the framework. We show the use of public domain tools, Doxygen, XML and Xerces-C++ for structural reflection of SystemC models, thus avoiding using front-end parsing tools such as EDG [49]. We exploit the open-source nature of SystemC to perform runtime reflection. We also introduce services for improved debugging, automated test generation, coverage monitoring, logging and a reflection service. ACE also provides design pattern implementations for multi-threaded models, thread pooling and synchronization. The inherent R/T facilities from TAO allows for R/T software modeling. This is also ideal for co-simulation, because it facilitates independent languages that support CORBA to easily interface with CARH. In CARH we have built a R/T middleware based infrastructure using Event service and Naming service among CORBA services, and by building a number of domain specific facilities such as reflection, automated test generation and dynamic-value change dump (d-VCD).

8.1 CARH's Capabilities

We enlist the fundamental contributions of our work:

- Service-orientation a necessary approach for allowing a multitude of features that enable multi-platform debugging, visualization, performance and coverage monitoring for system level designs.
- The use of a standardized interface-based mechanism for allowing the different services to communicate and exchange information amongst themselves. Using the OMG standardization of CORBA-variant implementations, any new feature that follows this standardization can be easily integrated into CARH.
- Introspective facilities for the models: Facilitating any such capabilities requires that the infrastructure have the ability to introspect the models. Our reflection and in-

[1]We code name our software systems after famous computer scientists. CARH (*kär*) stands for C. A. R. Hoare.

trospection mechanism improves debugging capability by providing automated test generation and extraction of runtime information of SystemC models.

- Debugging facilities: Unfortunately, standard debuggers prove less useful when debugging multi-threaded models and we need to facilitate the designer with model execution information in terms of call graphs, dynamic-Value Change Dump (d-VCD) and logging facilities.

- Automated test generation capabilities: Even though SCV has utilities for randomized constraint and unconstrained based test generation, there is no infrastructure that automatically generates testbenchs for particular models and stimulates them.

8.2 Issues and Inadequacies of Current SLDLs and Dynamic Validation Frameworks

There are numerous SLDLs employed in the industry such as Verilog, SystemVerilog, SpecC, VHDL and SystemC [12, 115, 13, 8] that primarily comprise of two aspects. The first being a modeling framework and the other being a simulation framework. The modeling framework allows designers to express designs either in a programmatic or graphical manner and the simulation framework is responsible for correctly simulating the expressed model. Designers can create models using the modeling framework and verify the design via simulation. However, with the rising complexity in current designs, it is not enough to simply provide designers with a modeling and simulation framework. It is becoming evident that designers require supporting tools to increase their productivity and reduce their design time. We enlist some of the important supporting features that are necessary for today's designers. They are: introspection in SLD languages and frameworks, automated testbench generation, coverage monitors, performance analysis and enhanced visualizations.

There are several industrial solutions that are targeting problem areas in SLDLs by providing some of these above mentioned features. However, instead of describing the multitude of commercial solutions for some of the inadequacies with SLDLs, we discuss some of the validation frameworks that are commercially available for improving design experience. Some of them consist of the features that we realize as being important for SLDLs.

Few of the popular validation framework solutions are ConvergenSC [22], VCS [116] and Cadence Incisive [24]. Each one of these tools have good solutions for different aspects in aiding designers for improved validation and model building experience. For example, ConvergenSC presents a SystemC integrated development environment (IDE) using Eclipse. This IDE has project management support, version control and build support. It also has an advanced debugging environment allowing step-wise execution that is specially geared towards QuickThreads used in SystemC. Although ConvergenSC provides a good IDE for improving design productivity and debugging, it does not support any testbench generation

features that are crucial for correctness of designs. On the other hand, VCS targets the aspect of RTL verification by incorporating multi-language support for Verilog, VHDL, SystemVerilog and SystemC, coverage techniques for Verilog and mixed-HDL designs, assertion-based design verification, and testbench generation constructs. It can interface with other Synopsys products. Similarly, Incisive from Cadence also supports some of the similar features for integrated transaction environment, unified simulation and debug environment, assertion support and testbench generation. By simply looking at some of these commercial solutions, we notice that neither one of these tools fully suffice the needs of a designer.

One apparent and a major drawback in the above mentioned industrial solutions is that of extendibility. Due to the commercial nature of these tools, it is impossible for designers to extend the existing framework themselves. Even though ConvergenSC's Eclipse IDE allows for plugins, their debugging facilities may not be easily extendible as desired by users. Furthermore, none of these solutions are open-source, disallowing users to consider alterations. Hence, designers have to look elsewhere for technology that can complement their existing solution. None of these tools can satisfy every designer's requirements thus necessitating the use of a variety of additional tools. For example, a designer may require ConvergenSC as an IDE, but also has to use VCS for RTL verification along with System Studio for system level design support. Even then, the use of multiple commercial tools may still not satisfy a specific purpose in the mind of the designer. This difficulty can be overcome by providing a framework that is a part of the public-domain, and that is easily extendible.

Another important concern is the deployment and reuse of technology. Often times, there are designers who create specific tools to perform certain tasks that are difficult to distribute because the tools are highly coupled with their environment. For example, suppose some designer implements a hot-spot analyzer for ConvergenSC as a plugin. This plugin would probably be highly coupled with the Eclipse and ConvergenSC environment making it difficult to adapt to other IDEs or environments. Therefore, another designer using a different environment may have difficulty in interfacing with this plugin without using that particular set of tools. Hence, it is important that extendibility is followed by a clean deployment mechanism so that plugins interact with their environment through an interface such that an easy adaptation to a different third party environment is possible. A well constructed solution for deployment also promotes unification of relevant tools with a clean interfacing mechanism that facilitates seamless interoperability between multiple tools.

8.3 Our Generic Approach to Addressing these Inadequacies

We propose a generic approach that addresses the primary inadequacies of extendibility, deployment and reuse in existing validation frameworks and features for supporting tools

that we perceive as being important for SLD languages and frameworks.

8.3.1 Service Orientation

Our approach promotes the idea of service orientation in SLDLs, where features are implemented as services that interact with each other through IDL interfaces that are language neutral interfaces defined by OMG [64]. Thus, the entire architecture comprises of multiple floating services that can be queried by clients through a common interface. Furthermore, this solution must be part of the public-domain such that designers may add services at their will.

Extendibility comes naturally with such an approach because a particular feature can simply be integrated into the architecture as a service that can also employ other existing services. In addition, deployment is relatively easy because the newly added service follows a strict interface convention that it must adhere. Lastly, the implementation of the feature can be independent of the interface as a service, allowing easy distribution and reuse of the feature.

Furthermore, features from different language paradigms may be integrated with the architecture as long as the language can interact with the interface mechanism. This means multi-language services can be integrated into the architecture through this service/interface mechanism. For example, a visual interface displaying performance graphs and tables may use Java as the core language. However, the actual simulation is performed in a C++ based environment and the interface mechanism allows for the two to communicate and exchange messages.

The advantage of following the OMG standardization, which TAO or any other CORBA-variant adheres to, is that all implementations of OMG standards can be seamlessly used with CARH. A commercial tool that can conform to such a standard allows for easy integration and adoption to CARH.

In addition, these features can be distributed over multiple computers or networks promoting a distributed simulation and validation infrastructure. An example of using a distributed framework is described in [68], that allows compilation and execution of testbenchs on a distributed network.

8.3.2 Introspection Architecture

The use of introspection is commonly seen in programming languages such as Java and languages that use the .NET framework. However, infiltrating introspective capabilities in SLDLs is still a sought after feature. Most attempts at introspection in SLDLs are facilitated via structural reflection, but runtime reflection also offers even more possibilities. If an SLDL inherits introspective capabilities, then projects such as IDEs, value-change dump

(VCD) viewers, improved debugging, and call graphs could take advantage of the reflected information. Unfortunately, there are very few non-intrusive methods for structural reflection and hardly any for runtime reflection [117].

8.3.3 Test Generation and Coverage Monitor

Most designers are also responsible for constructing their testbenchs that perform sanity and functional tests on their designs. Manual testbench generation is tedious and many times randomized testbenchs, weighted testbenchs, range-based testbenchs are sufficient in validating functional correctness. Furthermore, a testbench generator can take advantage of an introspective architecture to query information regarding ports, signals, types, bitwidths, etc. and then automatically with some user-hints generate testbenchs. However, most SLDLs do not come with a facility to automatically generate testbenchs. Thus, making it another important supporting feature for SLDLs.

Coverage monitors provide a measure of the completeness of a set of tests. Computing coverage during validation requires runtime information. This is where the runtime reflection capability can be exploited. In this paper we do not discuss about specific algorithms for test generation or coverage driven test generation because this paper is about the service oriented validation framework and these are but a few examples of services we need to integrate in such an environment. So the interfaces of these services are relevant to this paper and not the algorithms themselves.

8.3.4 Performance Analysis

As with most large designs, simulation efficiency is usually a major concern for designers for timely validation purposes. For this, performance analysis features are essential to SLDLs. There are designers who require hot-spot analysis to identify the bottlenecks of the design such that they can focus their optimization techniques towards that section. There are numerous metrics for measuring the time consuming blocks. For hardware design languages using a discrete-event simulation, we envision a performance analysis service that provides the designer with the following capabilities:

- The amount of time every hardware block or module consumes.
- The time spent in computation per module versus inter-module communication time.
- The number of times a particular block is scheduled to execute.
- The frequency of delta events and timed events generated by modules.
- Designer specified timers for identifying time taken in particular sections of the implementation.

- Version comparisons such that altered versions of the design can be aligned with the previous versions and performance metrics can be easily compared through graphs and tables.

These are some of the many capabilities that we see important for performance comparisons. However, performance analysis features are crucial in SLDLs for improving simulation efficiency of the designs, thus an important required feature for SLDLs.

8.3.5 Visualization

When working with large designs, a visual representation can be helpful in many ways. Visualization is an important tool for the designer to keep an overview and better manage the design. Using a graphical representation than requiring designers to traverse through many lines of code can immensely benefit design experience. Especially during architectural exploration and refinement, visualizations can help to take important decisions or to reveal problematic areas of the design. Visualizations for communication hot-spots, code usage, or online control flow analysis can give fast intuitive information about important key figures of the design.

In order to obtain meaningful and visually appealing graphs, we need two things: (i) The required data has to be collected and be made easily available. This can be rather obvious for static information such as the netlist or the module hierarchy, but may require important infrastructure support for dynamic information such as communication load or code coverage. (ii) The data has to be processed into visual graphs. This can be a very complex task depending on the type of graph and the level of desired flexibility and interaction. However there are many existing libraries and toolkits that can be used to render a graph structure. The Debussy nSchema and nWave modules [21] for example, are commercial tools for waveform and structural layout visualization. The GraphViz package [118] is an example for a comprehensive graph visualization package that can render and display different types of graphs such as simple textual description and others. Since SLD languages do not come with tools providing such visualizations, we see an important need to add this infrastructure to SLD toolkits in order to take advantage of advanced SLD features. Visualization in an SLDL toolkit can use data available from other services such as the coverage monitors, introspection architecture, and performance analysis and aid the designer in better comprehending this data. The more services an SLDL offers the more possibilities are there. Visualization can help to take design decisions based on more information in less time. The authors of [58] report an approach for interfacing visualization GUIs to SystemC. However, with the reflection capabilities in CARH we can provide a similar interface in an easier manner.

Until now we discussed the role of SLDLs in modeling and simulation, the need for supporting tools for SLDLs and a good infrastructure for deployment, extendibility and reuse. Now, we present CARH, a service oriented framework for validation of SystemC models. The

SLDLs we choose for our experimentation is SystemC [8] and employ the TAO [63] and ACE [61] libraries for the service orientation and implement additional services to promote the supporting features.

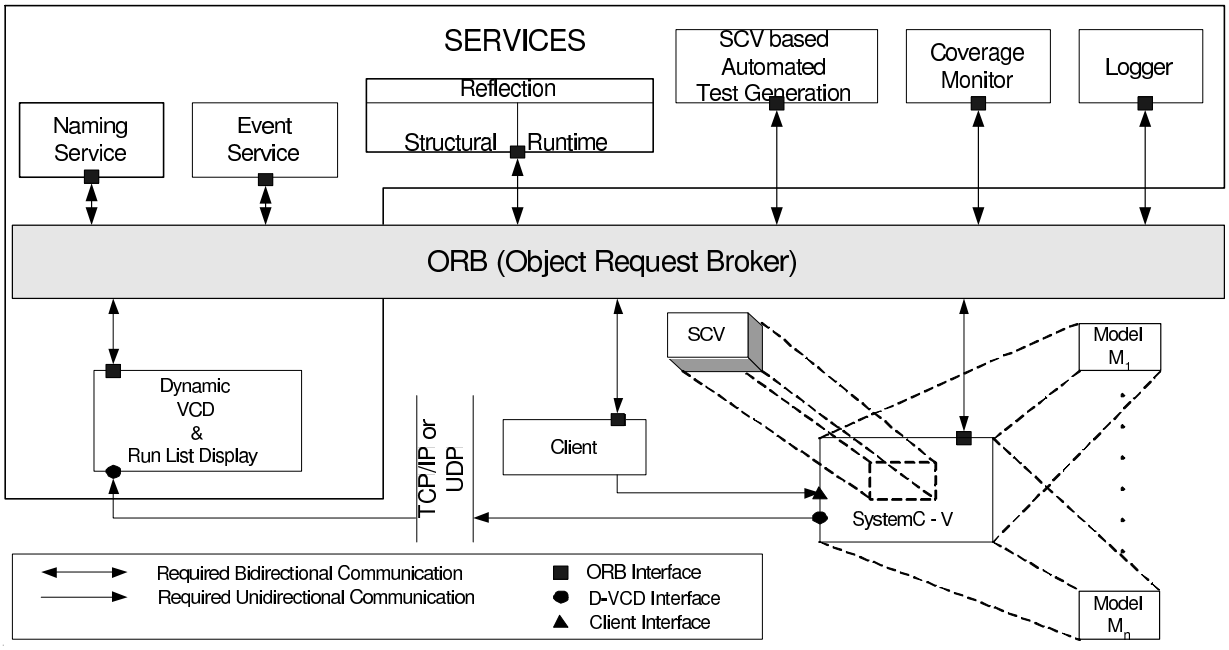


Figure 8.1: CARH Software Architecture

8.4 CARH's Software Architecture

The architectural description in Figure 8.1 serves as an extendible road map for CARH. The services and the SystemC models (SystemC-V) are the two main separations in Figure 8.1. The distinguishing arrows show interaction between the services and the SystemC model. The primary interaction is using an ORB to access services other than the d-VCD that happens to use TCP/IP protocols for communication. First we briefly describe the services within CARH:

Reflection: This consists of two sub-services, where the first exposes structural information about the model and similarly the second exposes runtime/behavioral information about the model.

Testbench: This service automatically generates SCV-based testbenchs for the SystemC model using the introspection capabilities provided by CARH.

Coverage: This performs coverage analysis on collected runtime information and results from simulation to regenerate better testbenchs.

Logger: It allows logging of runtime information based on logging requests entered through

the user console.

d-VCD: The d-VCD service is independent of an ORB and communicates through network communication protocols. The test generation, coverage monitor and logger services require the reflection and CORBA services. These services can be configured and invoked by a user console. It provides a dynamic-Value Change Dump on the reflected module along with the processes on the SystemC runlist.

The services below are employed by CARH but implemented in CORBA:

Naming: Allows a name to be associated with an object that can also be queried by other services to resolve that name and return the associated object.

Event: Allows decoupled communication between the requestors/clients and the services via an event based mechanism. A clean method to implement push and pull interactions between clients and services.

The remainder elements and facilities of the architecture shown in Figure 8.1 are described below:

SystemC-V: We implement SystemC-V that supports the SystemC Verification library (SCV) [8] and it also contains an extended version of SystemC 2.0.1 that communicates on an ORB and presents runtime information for introspective clients/services.

The ORB: The OMG CORBA [59] standard distinguishes between horizontal CORBA services, and vertical CORBA facilities. According to that terminology, only CORBA horizontal services we use from TAO are Event and Naming services. On the other hand the reflection, logger, test generation and coverage monitoring services are implemented by us, and qualify as domain specific vertical services.

Console: CARH is operated via the client. The client is a text-based interface that has commands to perform tasks for the user such as startup services, set specific service related flags, specify models to execute and so on.

One of the main strengths of CARH is the possibility of extensions. Developers can write TAO-based services and easily interact with the existing architecture via the interfaces defined. This opens up the possibility of an architecture that can be extended to support what is described in [65] and [68] along with many other pluggable services. Those services can easily be integrated in this architecture. Moreover ACE allows the use of design patterns [62] that can be effectively used to implement multi-threaded models and design-specific servers. For example, our implementation of d-VCD uses the Acceptor-Connector design pattern from the ACE library. The multiple models shown being executed on SystemC-V are a consequence of leveraging the ACE threading mechanisms to simulate concurrent models. The Usage model in Section 8.6 steps through showing how CARH is used. Obviously, there is a simulation performance price to pay for using such an infrastructure. Our early experiments show them in tolerable ranges as reported in Section 8.7.

8.5 Services Rendered by CARH

The ability to introspect structural characteristics of a model promotes a large variety of services, of which we currently implement the automated testbench generator and logger services. However, with emphasis on improving methodologies for debugging and model building experience, we extract runtime characteristics from the model as well and describe the d-VCD service. These two services made possible by the introspective architecture are not to be thought of as the only possible services, but simply two of the many that may follow. In addition, with the inherent facility of CORBA with the inclusion of TAO, we provide an elegant infrastructure for a distributed test environment. In this section, we describe these services in effort to prescribe examples which can employ the R-I and TAO capabilities.

8.5.1 Reflection Service

Doxygen, XML & Structural reflection: Processing C++/SystemC code through Doxygen to yield its XML documentation is ideal for data reflection purposes. The immediate advantages are that Doxygen output inserts XML tags where it recognizes constructs specific to the language such as C++ classes and it also preserves the source code line by line. The first advantage makes it easy to target particular class objects for further extraction and the latter allows for a well-defined medium for extracting any information from the source that may not be specific to an implementation. Since all SystemC constructs are not recognized during this pre-processing, we use the well-formed XML format of the source code as input to an XML parser to extract further structural information.

We leverage this well-formed XML-based Doxygen output to extract all necessary SystemC constructs not tagged by Doxygen itself using the Xerces parser and we implemented an additional C++ library to generate an Abstract System Level Description (ASLD). ASLD is written in a well-formed XML format that completely describes each module with the following information: signal and port names, signal and port types and bitwidths, embedded SystemC processes and their entry functions, sensitivity list parameters, and its hierarchy. We introduce a DTD specific for SystemC constructs that validates the ASLD for correctness of the extraction. The reflection service reads the ASLD and populates a data structure representing the model. Finally, TAO/CORBA client and server interfaces are written to make the reflection into a floating CORBA facility accessible via the interfaces. Our website [86] contains all the details and code.

Given an overview of our introspective architecture, we continue to present details on the infrastructure for introspection, with SystemC being the targeted SLDL of choice. We only provide small code snippets to present our approach and the concept of using Doxygen, XML, Xerces-C++, and C++ data structure to complete the reflection service. We present details of the Doxygen pre-processing, XML parsers employed in extracting uninterpreted

information from SystemC source files, our method of storing the reflected meta-data and our data structure allowing for introspection.

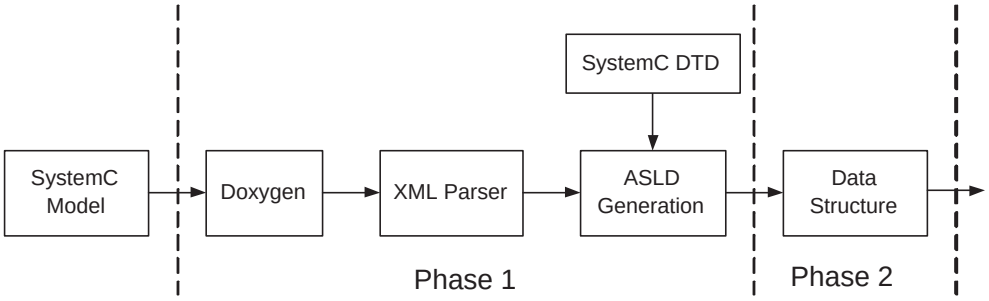


Figure 8.2: Design Flow for Reflection Service

Doxygen pre-processing: Using Doxygen has the benefit of simplifying C/C++ parsing and its corresponding XML representations. However, Doxygen requires declaration of all classes for them to be recognized. Since all SystemC constructs are either, global functions, classes, or macros, it is necessary to direct Doxygen to their declarations. For example, when Doxygen executes on just the SystemC model then declarations such as `sc_in` are not tagged, since it has no knowledge of the class `sc_in`. The immediate alternative is to process the entire SystemC source along with the model, but this is very inconvenient when only interested in reflecting characteristics of the SystemC model. However, Doxygen does not perform complete C/C++ compilation and grammar check and thus, it can potentially document incorrect C/C++ programs. We leverage this by adding the class definition in a file that is included during pre-processing and thus indicating the classes that need to be tagged. There are only a limited number of classes that are of interest and they can easily be declared so Doxygen recognizes them. As an example we describe how we enable Doxygen to tag the `sc_in`, `sc_out`, `sc_int` and `sc_uint` declarations. We include this description file every time we perform our pre-processing such that Doxygen recognizes the declared ports and data-types as classes. A segment of the file is shown in Figure 8.3, which shows declaration for input and output ports along with SystemC integer and SystemC unsigned integer data-types.

The resulting XML for one code line is shown in Figure 8.4. Doxygen itself also has some limitations though; it cannot completely tag all the constructs of SystemC without explicitly altering the source code, which we avoid doing. For example, the `SC_MODULE(arg)` macro defines a class specified by the argument `arg`. Since we do not include all SystemC files in the processing, Doxygen does not recognize this macro when we want it to recognize it as a class declaration for class `arg`. However, Doxygen allows for macro expansions during pre-processing. Hence, we insert a pre-processor macro as: `SC_MODULE(arg)=class arg: public sc_module` that allows Doxygen to recognize `arg` as a class derived from class `sc_module`. We define the pre-processor macro expansions in the Doxygen configuration file where the user indicates which files describe the SystemC model, where the XML output

```
/*! SystemC port classes !*/
template<class T> class sc_in { };
template<class T> class sc_out { };

/*! SystemC datatype classes !*/
template<class T> class sc_int {  };
template<class T> class sc_uint { };
```

Figure 8.3: Examples of Class Declarations

should be saved, what macros need to be run, etc. We provide a configuration file with the pre-processor macros defined such that the user only has to point to the directory with the SystemC model. More information regarding the Doxygen configuration is available at [52].

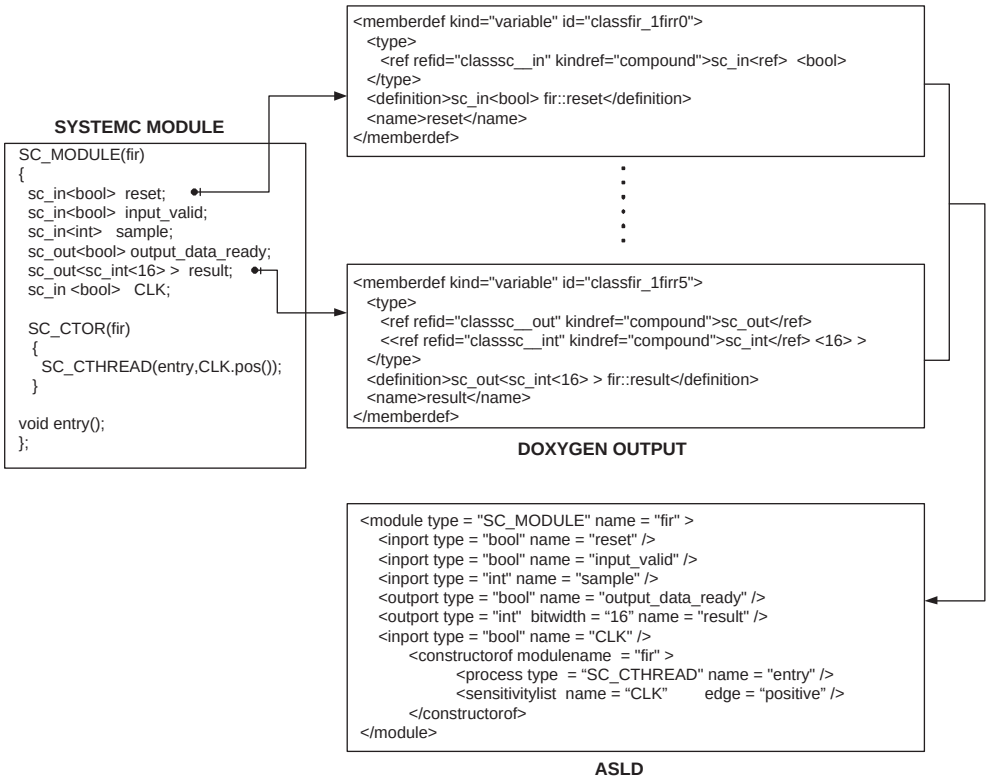


Figure 8.4: Doxygen XML Representation for `sc_in`

Even through macro pre-processing and class declarations, some SystemC constructs are not recognized without the original SystemC source code. However, the well-formed XML output allows us to use XML parsers to extract the untagged information. We employ Xerces-C++ XML parsers to parse the Doxygen XML output, but we do not present the source code here as it is simply a programming exercise, and point the readers at [119] for the source code.

XML Parsers: Using Doxygen and an XML parser we reflect the following structural characteristics of the SystemC model: port names, signal names, types and widths, module names and processes in modules and their entry functions. We reflect the sensitivity list of each module and the netlist describing the connections including structural hierarchy of the model stored in the ASLD. This ASLD validates against a Document Type Definition (DTD) which defines the legal building blocks of the ASLD that represents the structural information of a SystemC model. Some constraints the DTD enforces are that two ports of a module should have distinct names and all modules within a model should be unique. All these constraints ensure that the ASLD correctly represents an executable SystemC model. The main entities of the ASLD are shown in Listing 8.1.

ASLD: In Listing 8.1, the topmost *model* element corresponds to a SystemC model with multiple modules. Each *module* element acts as a container for the following: input ports, output ports, inout ports, signals and submodules. Each *submodule* in a *module* element is the instantiation of a module within another module. This way the ASLD embeds the structural hierarchy in the SystemC model and allows the introspective architecture to infer the toplevel module. The *submodule* is defined similar to a *module* with an additional attribute that is the instance name of the submodule. The *signal* element with its name, type and bitwidth attributes represents a signal in a module. Preserving hierarchy information is very important for correct structural representation. The element *inport* represents an input port for a module with respect to its type, bit width and name. Entities *outport* and *inoutport* represent the output and input-output port of a module. Line 16 describes the *constructorof* element, which contain multiple process elements and keeps a *sensitivitylist* element. The *process* element defines the entry function of a module by identifying whether it is an sc_method, sc_thread or sc_cthread. The *sensitivitylist* element registers each signal or port and the edge that a module is sensitive to as a *trigger* element. Connections between submodules can be found either in a module or in the *sc_main. Each* connection element holds the name of the local signal, the name of the connected instance and the connected port within that instance. This is similar to how the information is present in the SystemC source code and is sufficient to infer the netlist for the internal data structure.

Using our well-defined ASLD, any SystemC model can be translated into an XML based representation and furthermore models designed in other HDLs such as VHDL or Verilog can be translated to represent synonymous SystemC models by mapping them to the ASLD. This offers the advantage that given a translation scheme from say a Verilog design to the ASLD, we can introspect information about the Verilog model as well.

Listing 8.1: Main Entities of the DTD

```
1 <!ELEMENT model (module)* >
2 <!ATTLIST model name CDATA #REQUIRED>
3
4 <!ELEMENT module (inport | outport | inoutport | signal | submodule)* >
5 <!ATTLIST module name CDATA #REQUIRED type CDATA #REQUIRED >
6
7 <!ELEMENT submodule EMPTY >
8 <!ATTLIST submodule type CDATA #REQUIRED name CDATA #REQUIRED instancename CDATA #
```

```
      REQUIRED >
 9
10 <!ELEMENT signal EMPTY >
11 <!ATTLIST signal type CDATA #REQUIRED bitwidth CDATA #IMPLIED name CDATA #REQUIRED >
12
13 <!ELEMENT inport EMPTY >
14 <!ATTLIST inport type CDATA #REQUIRED bitwidth CDATA #IMPLIED name CDATA #REQUIRED >
15
16 <!ELEMENT constructorof (process * | sensitivitylist) >
17 <!ATTLIST constructorof modulename CDATA #REQUIRED >
18
19 <!ELEMENT process EMPTY >
20 <!ATTLIST process type CDATA #REQUIRED name CDATA #REQUIRED >
21
22 <!ELEMENT sensitivitylist (trigger)* >
23
24 <!ELEMENT trigger EMPTY >
25 <!ATTLIST trigger name CDATA #REQUIRED edge CDATA #REQUIRED>
26
27 <!ELEMENT connection EMPTY>
28 <!ATTLIST connection instance CDATA #REQUIRED member CDATA #REQUIRED local_signal
      CDATA #REQUIRED>
```

Data Structure: The ASLD serves as an information base for our introspection capabilities. We create an internal data structure that reads in this information, enhances it and makes it easily accessible. The class diagram in Figure 8.5 gives an overview of the data structure. The *topmodule* represents the toplevel module from where we can navigate through the whole application. It holds a list of module instances and a list of connections. Each connection has one read port and one or more write ports. The whole data structure is modeled quite close to the actual structure of SystemC source code. All information about ports and signals and connections are in the module structure and only replicated once. Each time a module is instantiated a *moduleinstance* is created that holds a pointer to its corresponding module.

The information present in the ASLD and the data structure does not contain any behavioral details about the SystemC model at this time, it merely gives a control perspective of the system. It makes any control flow analysis and optimizations on the underlying SystemC very accessible.

Focus-defocusing of models and modules: The reflection service also implements the idea of *focus*ing on modules. Since a model generally consists of multiple SystemC modules, the reflection service reflects information regarding the module in *focus*. This *focus* can be changed through interface functions accessible to the user. Furthermore, there can be multiple instances of the same or different model that needs reflecting. Similar to *focus*ing on modules, we implement *focus* support for models.

The source code for this introspective infrastructure in SystemC is available online at [119] for download.

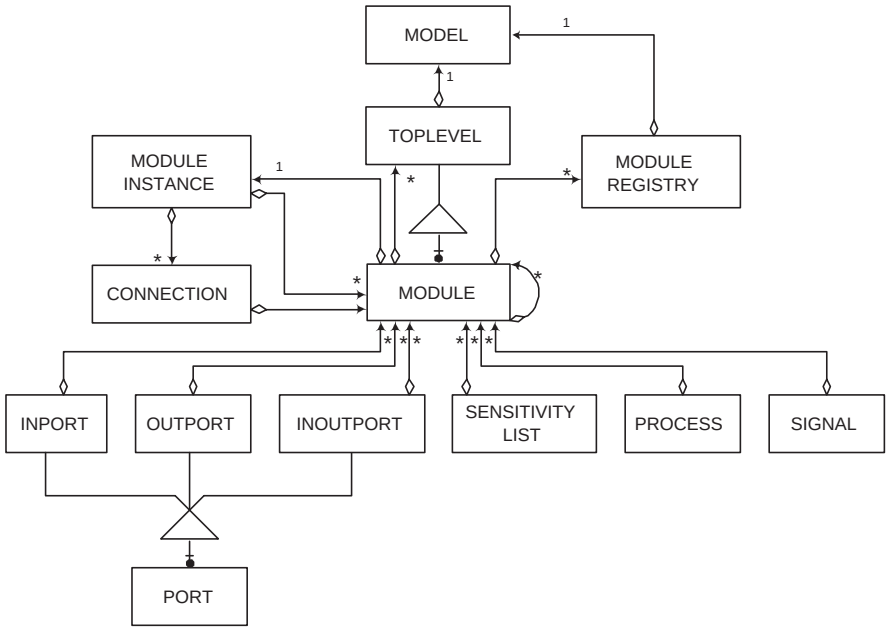


Figure 8.5: Class Diagram Showing Data Structure

8.5.2 Testbench Generator

CARH provides automated test generation and coverage monitoring as vertical services for testing SystemC models. The test generation service is built using the SystemC Verification (SCV) [8], which is a library of C++ classes, that provide tightly integrated verification capabilities within SystemC. This service takes the name of the model, a data file and a few user specified parameters to generate customized testbenchs for the model.

The client issues commands that specify the model and module in *focus* to the test generation service, which invokes the respective API call on the reflection object that creates the corresponding ASLD for the SystemC model. Then it invokes the API call for initialization of the data structure and enabling the introspective capabilities of the reflection object. This allows the test generator to introspect the reflected information and automatically generate a testbench based on the user-defined data file and parameters. The test generator searches for marked ports for the given module and introspects them. If none were found, then all the ports for the given module are introspected and a corresponding testbench is generated.

The test generator can create constrained and unconstrained randomized testbenchs. In the *unconstRand mode*, unconstrained randomized testbenchs are created, which use objects of *scv_smart*
_prt<T> type from SCV. This is the default mode of the test generator. In the *simpleRand mode*, constrained randomized testbenchs are created. These testbenchs issue *Keep_out* and *Keep_only* commands to define the legal range of values given by the user in the data file. Similarly in the *distRand mode*, *scv_bag* objects are used in testbenchs given the appropriate

Algorithm 4 Test Generation

```
1: Invoke generate_moduletb(module m, option p, datafile d)
2: if marked ports exist then
3:   For each marked port 'pt'
4:     if 'pt' exists in module 'm' then
5:       if p = "unconstRand" then
6:         Query reflection service for type of port 'pt'
7:         Query reflection service for bitwidth of port 'pt'
8:         Generate randomized testbench
9:       else if p = "simpleRand" then
10:        Repeat Step 2 & 3
11:        Generate simple constrained randomized testbench
12:      else if p = "distRand" then
13:        Repeat Step 2 & 3
14:        Generate randomized testbench with distribution modes
15:      end if
16:    else
17:      Print "Test Generation Failed"
18:    endif
19: else
20:   For all ports in module 'm'
21:     Repeat through Steps 4 to 18
22: end if
```

commands from the console and providing the data file with the values and their probability. Figure 8.6 and 8.9 show snippets of executing the FIR example using CARH.

Testbench generation Example

We briefly describe the testbenchs generated using the FIR example from SystemC distribution. In particular, we set focus on the computation block of the FIR. We present Figure 8.6 that shows three testbenchs using the *unconstRand*, *simpleRand* and *distRand* modes. The *unconstRand* generates unconstrained randomized testbenchs, the *simpleRand* constrain the randomization using *keep_out* and *keep_only* constructs with legal ranges specified from an input data file and the *distRand* defines *SCV_bags* that give a probabilistic distribution for the randomization. Once, the automated testbench generated, it is integrated and compiled to test the FIR block. The integration is performed manually by defining the appropriate interface between the generated testbench and the FIR block.

```
/*! Defining an SCV smart pointer !*/
scv_smart_ptr <int> r_sample;
/*! Generating the randomized values !*/
r_sample->next();
```

Snippet of the testbench in the *unconstRand* mode

```
/*! Defining simple constraints !*/
scv_smart_ptr <int> r_sample;
/*! Defining the legal ranges !*/
r_sample->keep_only ( 10,100 );
r_sample->keep_out ( 21, 49 );
r_sample->keep_out ( 61, 89 );
```

Snippet of the testbench in the *simpleRand* mode

```
/*! Defining weights for the distribution mode !*/
scv_smart_ptr <int> r_sample;
scv_bag<pair<int,int> > d_sample;
/*! Defining the legal ranges !*/
d_sample.add(pair<int, int> (1, 3), 40 );
d_sample.add(pair<int, int> (5, 7), 30 );
/*! Setting the distribution mode !*/
r_sample->set_mode(d_sample);
```

Snippet of the testbench in the *distRand* mode

Figure 8.6: Code Snippets for Generated Testbenchs

We intend to improve our automated testbench generation capabilities by first implementing additional services such as coverage monitors and simulation performance monitors to better analyze the SystemC model. These additional services will assist the testbench generator in making more intelligent and concentrated testbenchs.

Distributed Test Environment: Currently, the user compiles the testbench with the model through the console. However, we target CARH to handle compilation and execution as described in [68] and shown in Figure 8.7. This allows the testbenchs to compile and execute independently through interface calls on the ORB. In effect, this distributed test environment also supports co-simulation where different languages that support CORBA

interfaces may communicate through the ORB. Testbenchs could be written in languages different from regular HDLs as long as they interface correctly with the ORB. Furthermore, visualization tools interfacing through the ORB can greatly benefit from such an architecture. Pseudocode 4 shows steps involved in generating a testbench for a module and the interaction between the test generation and reflection service.

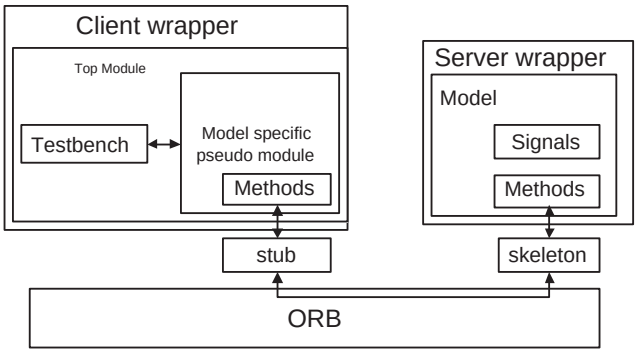


Figure 8.7: Test Environment

8.5.3 d-VCD Service

Implementing runtime reflection mandates alterations to the existing SystemC source. This is unavoidable if runtime information has to be exposed for SystemC and we justify this change by having two versions of SystemC. We call our altered version SystemC-V, which the designer can use for the purpose of verification and debugging of SystemC models. However, for fast simulation the same model can be compiled with the original unaltered version of SystemC by simply altering the library target in the Makefiles.

The d-VCD service displays signal value changes for a module "as they happen". Regular VCD viewers display VCD information from a file generated by the simulation. However, we enable the d-VCD viewer to update itself as the signals of a focused module in the SystemC model changes. Every signal value change for the module in focus communicates with the d-VCD. Likewise, at every delta cycle we send the process names on the runlist to the d-VCD. Figure 8.8 shows a screenshot of a GUI for the VCD using Qt [120]. To enable SystemC-V to expose this information we altered the `sc_signal` class along with adding an extra class. Before discussing brief implementation details it is necessary to understand how we utilize the reflection service. In order to gain access to the reflected information, we instantiate an object of class `module` and use it as our introspective mechanism. Member functions are invoked on the instance of `module` to set the focus on the appropriate module and to introspect the characteristics of the module.

To facilitate SystemC for exposing runtime characteristics, we implement class `fas_sc_signal_info` that stores the signal name (`sig_name`), signal type (`sig_type`) and classifica-

tion type (`sig_class`) with their respective set and get member functions. SystemC has three class representations for `sc_signal`, where the first one is of template type `T`, the second is of type `bool` and the third is of type `sc_logic`. Each of these classes inherit the `fas_sc_signal_info` class minimizing changes to the original source. In fact, the only changes in the original source are in the constructor and the `update()` member functions. We require the user to use the explicit constructor of the `sc_signal` class such that the name of the signal variable is the same as the parameter specified in the constructor. This is necessary such that an object of `sc_signal` concurs with the introspected information from the reflection service. We also provide a PERL script that automatically does this. The `update()` function is responsible for generating SystemC events when there is a change in the signal value and an ideal place to transmit the data to the d-VCD service. So, if the signal type is a SystemC type then the `to_string()` converts to a `string` but if it is classified as a C++ type then it is converted using `stringstream` conversions.

The explicit constructors invoke `classify_type()` which classifies the signal into either a SystemC type or a C++ type. We use the classification to convert all C++ and SystemC types values to a `string` type. This is such that multiple VCD viewers can easily interface with the values returned from SystemC-V and they need not be aware of language specific datatypes. Since all SystemC datatypes have a `to_string()` member function, it is easy to return the string equivalent for the value. However, for C++ data-types we employ a work around using `stringstream` conversion to return the string equivalent. Even though, we are successfully able to translate any native C++ and SystemC data-types to their string equivalent, the compilation fails when a SystemC model uses signals of C++ and SystemC types together. This is because for C++ data-types the compiler cannot locate a defined `to_string()` member function. An immediate solution to this involves implementing a templated container class for the templated variables in `sc_signal` class such that the container class has a defined function `to_string()` that allows correct compilation. We add the class `containerT<T>` as a container class and replace variable instances of type `T` to `containerT<T>` in order to circumvent the compilation problem. We interface the runtime VCD with the Qt VCD viewer implemented by us, shown in Figure 8.8.

With dynamic runtime support in SystemC, we also expose runlist information as another example of runtime reflection. Being able to see the processes on the process runlists gives a significant advantage during debugging. This capability is available by altering the SystemC `sc_simcontext` class. Figure 8.8 also shows the output for the processes on the runlist along with the dynamic value changes on signals. Exposing the process name to the d-VCD service itself does not require the reflection service since the process names are available in SystemC `sc_module` class via the `name()` member function. However, using the reflection service we provide the user with more concentrated visuals of model execution by enabling the user to specify which particular module's processes should be displayed. This requires querying the reflection service for the name of the modules in focus and only returning the names of the processes that are contained within those modules. Implementing this capability required stepping through SystemC runlist queues and monitoring whether they match the modules

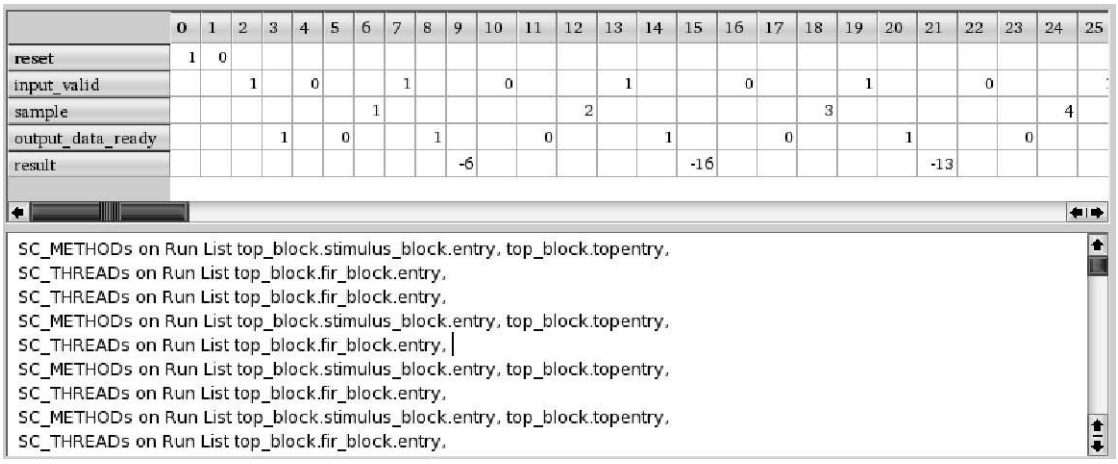


Figure 8.8: d-VCD Output

of interest and transmitting the name to the d-VCD service.

8.6 Usage Model of CARH

User Console: We developed an interactive shell for CARH, which is shown as Client in Figure 8.1. The user initiates the shell and interacts with it to setup the services, request for a testbench, and execute testbench and model. Figure 8.10 shows an interaction diagram with the complete usage model. The usage model starts at the point where the user needs verification of a SystemC model and initiates the interactive shell to do so as shown in Figure 8.9.

Load configuration: The first step is to *load the configuration file* using the `load_config <path>` command. This loads the system specific paths to the required libraries and executables.

Specify model: Assuming that the configuration paths are correct, the user invokes the `load_model <path> <modelname>` command to specify the directory of the model under investigation and a unique `modelname` associating it. This command flattens the model into a single flat file by concatenating all `*.h` and `*.cpp` files in the specified directory followed by running Doxygen on this file. This results in a well-formed XML representation of the SystemC model. For an example see the FIR model in Figure 8.9 that shows snippets of the Doxygen output and the ASLD generated.

Initiate services: To start up CARH's services, the user must invoke `startup_services` followed by `init_clients`. These two commands first start the reflection service followed by the d-VCD and then the automated test generator.

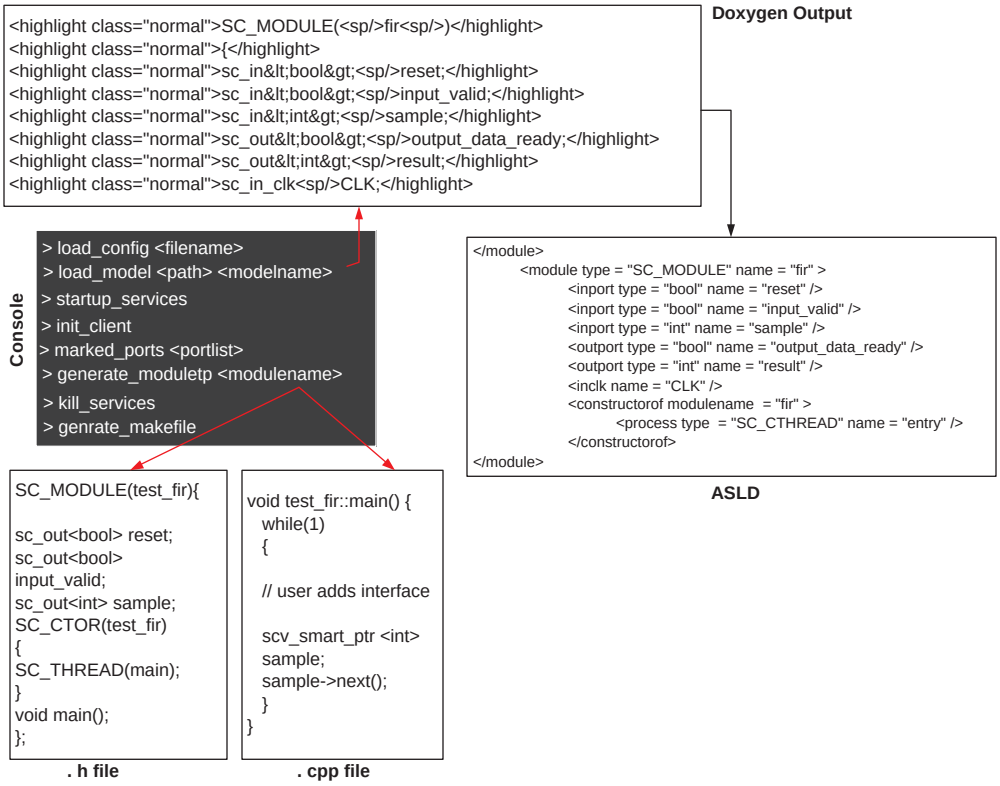


Figure 8.9: Snippets of Intermediate Files

Mark ports: Ports can be marked that are given as input to the test generator. The user can also create a file with the listing of the module names and the ports of interest and load these settings. However, the command `marked_ports <portlist>` initializes the ports in `portlist` for introspection and test generation focusing on these ports.

Test generation: The users can then request testbenchs by calling `generate_module-tb<module_name>` or `generate_topleveltb`. We only present the default test generation commands in this usage model. The `generate_moduletb` creates a testbench for the specific `module_name`. However, the user can use `generate_topleveltb` command to generate a testbench for the entire model, except that this requires wrapping the entire model in a toplevel module. We require the user to wrap it in a module called *systemc_data_introspection* to indicate to the reflection service the toplevel module. The user can also request for the Coverage Monitor service for the appropriate test. Figure 8.9 also shows the .h and the .cpp file generated for the computation module of the FIR.

Behavioral information: After successful testbench generation, the user needs to add interface information into the test such that it can be integrated with the model. Consider the computation module of the FIR, the interfacing information would be regarding how many cycles the reset needs to be applied and when the inputs need to be sampled. Such

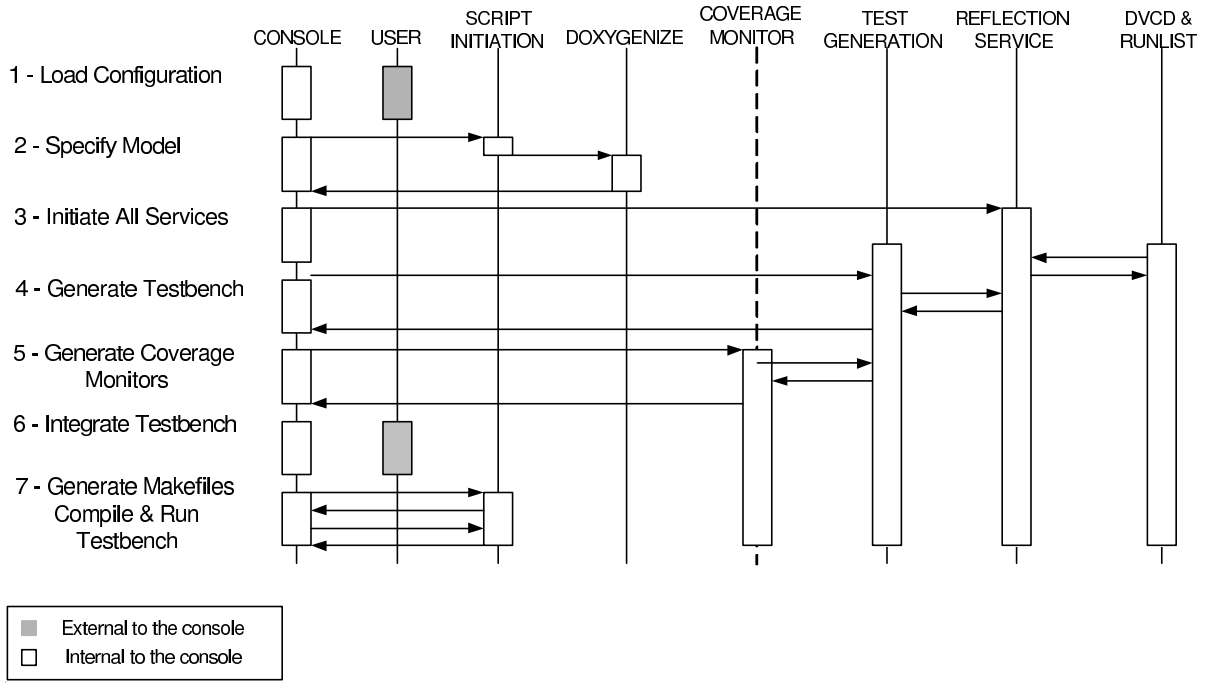


Figure 8.10: CARH's Interaction Diagram

behavioral aspects are not yet automated with our test generation service. Then, the user can use the `generate_makefile` command to produce *Makefile*s specific for the model after which the testbench can be compiled and executed from the console. While the model executes, the d-VCD service does a dump of the different value change that occur across the various signals of the model. A screenshot of the d-VCD for the FIR example is shown in Figure 8.8. It also displays the processes on the runlist.

```
SYSTEMC=/home/deepak/sc-ace-rdy-2.0.1
ACE_ROOT=/home/deepak/ace/ACE_wrappers
TAO_ROOT=/home/deepak/ace/ACE_wrappers/TAO DOXY_ROOT=/usr/bin
REFLECTION_SVC=/home/deepak/ace/ACE_wrappers/Reflection
NAMING_SVC=NameService
TESTGEN_SVC=/home/deepak/ace/ACE_wrappers/Reflection/tgn
DVCD_SVC=/home/deepak/ace/ACE_wrappers/Reflection/dvcd
```

Figure 8.11: Snapshot of Configuration File

	FIR (seconds)		FFT (seconds)	
Samples	Original	CARH	Original	CARH
100000	4.16	189.42	2.91	191.17
200000	8.20	384.02	5.81	377.47
300000	12.41	566.25	8.79	586.58
400000	16.54	757.02	11.70	788.60
500000	20.67	943.23	14.62	984.03

Table 8.1: Simulation Results on FIR & FFT

8.7 Simulation Results

Table 8.1 shows simulation times in seconds for two examples: Finite Impulse Response (FIR) filter and Fast Fourier Transform models. Columns marked *Original* refers to the model with the testbench generated from the test generation service compiled with SystemC 2.0.1 and SCV version 1.0p1 and CARH refers to the testbench compiled with SystemC-V and CARH infrastructure. There is a performance degradation of approximately 45x and 65x for the FIR and FFT models respectively. This performance decrease is not surprising when using TAO. There is a significant overhead in communicating through an ORB as each of the services are spawned as separate processes registered with the ORB that require communication through CORBA interfaces. However, the facilities provided by CARH justifies the performance degradation because CARH offers multiple services and possibilities for various extensions. CARH can be used during development of the model for debugging, testing and so on, and when a simulation needs to be faster then the model can be easily linked to original SystemC and SCV libraries since no changes in the source are made. In fact, since services can be turned on and off at the designers will, we employed the R-I and testbench generation service to create our testbenchs for the FIR and FFT examples, after which we simply compiled and executed it with the unaltered version of SystemC. The simulation times were on average the same as the Original timings shown in Table 8.1.

Another important point to note is that TAO is just one implementation of CORBA and not the best stripped down version suitable for CARH. For example, it has real-time capabilities that we do not employ at all. We only use TAO as a solution to display our proof of concept for service orientation and industrial strength solutions could use a much lighter and smaller CORBA-variant. This would impact the simulation results directly as well.

8.8 Our Experience with CARH

The foremost important experience we discuss here involves adding a service to CARH. We recount briefly the steps in integrating this service without getting into the details of the algorithms employed. We also briefly describe our debugging experience with the d-VCD

and R-I services. This is followed by a table that describes some of the features CARH possess versus existing commercial tools.

So, for example, determining the number of times each SystemC process triggers could be a coverage metric sought after. Evidently, this requires runtime reflection that the R-I service provides. We implement the coverage service in C++ (or any CORBA-compliant language) and begin by constructing a data structure that can store the SystemC process names and an integer value representing the trigger count. We define the appropriate member functions that can be used to populate this data structure. This first step of implementing the data-structure and its API in C++ is a programmer dependent task regarding the time required to implement. For us, it took us less than an hour. Until this stage we are only preparing the coverage monitor implementation and the following step involves wrapping this C++ implementation of the coverage monitor with CORBA service methods. However, before this, the IDL must be defined through which this coverage monitor service interacts with the other clients and services across the ORB. This step is almost similar to specifying the API of the data structure, since most of the member functions of the coverage monitor should be accessible via the ORB. Hence, the IDL construction itself requires little time. The third stage involves instantiating an ORB through which the service registers itself with the Naming Service allowing other services to locate this new service. We provide a simple wrapper class that allows easy integration of any generic C++ implementation into a service making this third step requiring only minutes. Finally, the implementation for requesting information from the R-I service through the ORB and populating the data structure is added to the coverage monitor. This completes the integration of our example of a basic monitor service into CARH. The entire process takes only a few hours, depending on the complexity of the actual core of the service. For services that require more complex data structures, algorithms etc., the most of the integration time is spent in programming these algorithms and not the integration. Furthermore, if there are existing libraries of algorithms, they can be easily used to provide the core technology / algorithms requiring the programmer to only write the IDL and wrapping the implementation into a service. The services we experiment with are mainly implemented by ourselves. We do not integrate a commercial tool into CARH as of yet.

Our debugging experience is limited to using the R-I and d-VCD services. We found that these two services did indeed help us reduce the debugging time, especially the display of processes being triggered. One interesting problem we discovered with our model using these services is a common one of missed immediate notification of an `sc_event`. In SystemC 2.0.1, there is no notion of an event queue for `sc_event`s added into the model for synchronization purposes. This is rectified in SystemC 2.1 with the introduction of an event queue such that no notifications are missed, but instead queued. However, with SystemC 2.0.1 (the version we use for our development) we were able to pinpoint and freeze the simulation just before the notification and then just after, to see which processes were triggered following the immediate notification. However, this is simply one experience of locating one bug. We understand that adding other debugging facilities will greatly improve locating design errors.

Capability	SystemC Studio	ConvergenceSC	Incisive	CARH
Extendibility	No	No	No	Yes
Interoperability	No	No	No	Yes
Services				
Reflection	No	No	No	Yes
Introspection	No	No	No	Yes
Test Generation	Yes	No	Yes	Yes
Coverage Monitors	Yes	No	Yes	No
Debugging (Call graphs, Execution traces, etc.)	No	Yes	Yes	Yes

Table 8.2: Brief Comparison between Commercial Tools and CARH

We present a table displaying some of the noticeable features in commercial tools compared with CARH. We base our comparison only on publicly available information and do not claim to know the details of the underlying technologies used in these commercial tools. Table II shows some of the feature comparisons.

In summary, CARH presents a methodology where we employ the use of public-domain tools such as Doxygen, Apache's Xerces-C++, TAO, and ACE to present a service oriented validation architecture for SystemC. In our case, we chose our SLDL to be SystemC, however that only serves as an example. We describe our approach in detail and also present CARH whose core feature is the introspective architecture. Services such as the automated test generator and d-VCD are some of the examples that utilize this R-I capability in SystemC. The use of services and the ORB suggests a tidy deployment strategy for extensions to a framework. Furthermore, using a CORBA-variant needs the extensions to adhere a strict interface that can easily be added as wrappers. This allows the design of the actual feature to remain isolated from the interface and can communicate through messages promoting reuse of technology. Even though there is a performance degradation in simulation results, the exercise of building highly complex models can be eased. For fast simulation, once the models are tested and verified, can be easily linked to the original SystemC libraries. Since, the services can be turned off and on at the designer's will, the simulation times may not be affected at all if say only the testbench generator service is employed. The main overhead comes in when using SystemC-V which requires exposing the internal characteristics of the model under test via an ORB. The simulation experiments showed that having all the services up induces a significant performance penalty. The performance of a simulation run in this framework is not optimized because we want to create a "proof of concept" and hence we used an existing CORBA implementation which is optimized for real-time middleware applications, and hence not necessarily customized for this application. However, we believe that if this idea catches on, we will create customized middleware compliant with OMG CORBA specifications that will be optimized for this specific purpose and hence we will have much better performance. However, to show the validity of such a plug-n-play validation

framework and infrastructure, we did not feel the need to demonstrate performance efficiency but rather show the viability of the implementation and illustrate the advantages.

The one most fundamental issue exposed in this work is of service-orientation and using a standardized specification for providing an architecture for validating system level models. With the use of the OMG specification and its implementation of the CORBA-variant TAO, we show the advantage of easy deployment, integration and distribution of features as services. Furthermore, due to the OMG standardization, any implementation abiding by this standardization would be easy to integrate into CARH. The ease of extendibility is natural with such an architecture. Having said that, CARH is a proof of concept for promoting a service oriented architecture and for better performance results, a light-weight CORBA-variant other than TAO should be used. On the other hand, CARH's extendibility can result in numerous useful tools for debugging, visualization, performance monitoring, etc. We simply show some of the possibilities to further promote the benefits of a service oriented architecture.

Our experience with CARH suggests an improved model building experience. In particular, automated testbench generation overcomes the need to create basic sanity test cases to verify the correctness of the design. Even with the basic automated testbench generation service, we could easily locate problematic test cases resolves as bug fixes in our designs. With the addition of more intelligent testbench generation algorithms using information from the coverage analysis service, we foresee a much improved test environment. As for the d-VCD, the process list visualization and the value changes are definitely useful. However, we understand that the need for step-wise execution of the model is crucial and the ability to pause the simulation is needed. We plan to implement this using the push-pull event service of TAO such that designers can step through event notifies and module executions.

Part of this work, the SystemC parser, is available as an open-source project called SystemCXML [119]. We also implement an automated test generation service that uses the existing SCV library to automatically generate testbenchs for SystemC models in CARH. In addition to that, we offer dynamic representation of the value changes shown by introducing the d-VCD service along with process list information. We implement a console through which a user controls this entire framework. We briefly present a list to summarize the features that we have implemented so far in CARH. (i) Reflection service provides structural and runtime information. (ii) Test generation service supports constrained and unconstrained randomized testbenchs using information from the reflection service. (iii) Naming service used to access all other services on an ORB. (iv) d-VCD service receives signal changes and runlist information. This uses an Acceptor-Connector design pattern from ACE. (v) Client console allows integration of all the services.

Chapter 9

Summary Evaluations

In this section, we evaluate our response to the questions we posed in Chapter 1. These questions address the aspects of modeling, simulation, validation and dynamic integration of multiple tools of an ESL methodology.

9.1 Modeling and Simulating Heterogeneous Behaviors in SystemC

The first question we asked ourselves was "How can we both model and accurately simulate heterogeneous behaviors in SystemC so that we can represent the inherent heterogeneity in designs, and the hierarchical embeddings of these heterogeneous behaviors?" We approached this question by first discovering the methods used in describing heterogeneous behaviors with standard SystemC. That is, using the inherent Discrete-Event MoC of SystemC to mimic the intended heterogeneous behavior. This is achieved by explicitly describing the heterogeneous behavior by synchronizing events [32, 15]. The difficulty in impersonating this heterogeneity is that engineers have to come up with a correct representation for the intended heterogeneity, which is not always intuitive or always correctly replicable. So, what we provide are heterogeneous kernel-level extensions to SystemC that implement the execution semantics of certain specific MoCs. Mainly, the Synchronous Data Flow (SDF) [33], Communicating Sequential Processes (CSP) [32], Finite State Machine (FSM) [29, 121, 32] and Bluespec's rule-based MoCs [103, 97]. These MoCs extend the modeling language of SystemC with constructs that allow engineers to explicitly describe designs using these MoCs. We select these MoCs because they represent the commonly seen behaviors in system-on-chip (SoC) designs. We present few of the possible MoCs as a proof of concept in representing heterogeneous behaviors in SystemC.

By endowing SystemC with these MoCs, we are able to represent data flow components

of SoCs as SDF models, controllers as FSMs and resource sharing components using Bluespec's rule-based MoC. This brings forth the advantage that engineers no longer have to use SystemC's Discrete-Event (DE) capabilities to mimic the heterogeneous behaviors. Instead, when a component's behavior is recognized as one of the MoCs, it can be modeled using that MoC's modeling language. Thereby, decomposing the intended design based on their inherent behaviors and representing them as models following MoCs.

The extended modeling language supports hierarchical design composition. For example, an FSM may embed another FSM within either its state or transitions. The support for hierarchical composition of designs is also supported for the SDF MoC. We went further and provided support for hierarchical composition across the SDF and FSM MoCs. This is an example of heterogeneous behavioral hierarchy, where an FSM may embed an SDF component and vice versa up to any arbitrary level of depth. This is an added benefit because certain large designs contain components that may have subcomponents that follow different MoCs.

The SDF and FSM MoCs display our proof of concept for supporting heterogeneous behavioral hierarchy with SystemC. However, note that our solution is limited in that heterogeneous behavioral hierarchy is not supported for the SystemC's DE and BS-ESL MoCs. For a better understanding of the reasons behind this, we must first discuss our solution to the simulation of heterogeneous behavioral hierarchy. It is straight forward to see that extending the modeling language of SystemC must have its counter part simulation framework. So, we implement the execution semantics of each of the MoCs in our kernel extensions. This allows engineers to simulate their designs described using the extended MoCs. They can also embed behaviors of the same MoC within executable entities (such as states and transitions for FSM MoC and function blocks for SDF) that we call behavioral hierarchy. This makes it natural for engineers to describe their models in the same manner in which the design is composed. So, for example, if a controller design contains a state that has other FSMs embedded in them, then we can represent this in the exact same manner in our model. The execution semantics of the MoC must support this sort of hierarchy for simulation. We additionally explore heterogeneous behavioral hierarchy by introducing execution semantics that support the hierarchical embedding of heterogeneous models with the SDF and FSM (both hierarchical versions of the MoCs). The advantage of this is clear. Suppose that a function block of a data flow design contains a controller component, then it can be naturally represented and simulated using the extensions. This takes us back to why we do not support heterogeneous behavioral hierarchy for the DE and Bluespec's rule-based MoCs. For SystemC's DE MoC, we can interoperate the extended MoCs with SystemC being the master (we call this the kernel that drives other kernels), but a SystemC component cannot be embedded in say an SDF function block. Such a capability requires altering the reference implementation of SystemC such that different DE simulation kernels can be invoked for every level of hierarchy as per our execution semantics for heterogeneous behavioral hierarchy (described in Chapter 4 and Chapter 5). However, we do not want to alter the reference implementation, thus disallowing us to provide this capability for the DE MoC. This same reasoning holds for Bluespec's

rule-based MoC. Since it was accepted as an industrial product, we did not want to alter the original reference implementation. Another reason is that descriptions using Bluespec's rule-based MoC should have a corresponding hardware representation. This means that it should be synthesizable. Our focus in this dissertation is not on the synthesis aspect of an ESL methodology, though it is certainly possible and left aside for future work.

9.2 Validating Corner Case Scenarios for SystemC

The next question we asked ourselves refers to the validation aspect of an ESL methodology and this question was "How can we generate input sequences for validating corner case scenarios of a SystemC implementation by using an abstract representation to explore the possible input sequences and then using these input sequences to drive the implementation model?" Our solution is based on the Abstract State Machines [79, 78] and the SpecExplorer tool [80]. Using these tools, we show how to formally describe designs in the AsmL and use exploration facilities of SpecExplorer to create an automaton for the semantic model. Then, we generate tests using the test generation facility of SpecExplorer. After which, we create a set of wrappers to tie together an implementation model in SystemC with the semantic model in AsmL such that inputs for the tests in the semantic model simultaneously drive the inputs of the implementation. Our approach to the solution has been a model-driven approach that involves creating the semantic model then the implementation model as a refinement of the semantic model [122].

We recognize the problem faced by engineers as not having any systematic method for generating input test sequences for corner case scenarios of the system. We introduce the facility of being able to explore an abstract representation (semantic model) for generating these input sequences. This systematic method is the exploration of the semantic model, which mandates using the semantic model for creating an automaton representing the particular test scenario. This is done by indicating the states of interest, the transitions that update values on these states and the desired final states. The resulting automaton is then traversed to yield test sequences. However, this only creates tests for the semantic model and we want to create tests for the implementation model. Since the implementation model is a refinement of the semantic model, any state element present in the semantic model must have a corresponding state element in the implementation model. This holds true for the transition functions as well. Given that we can generate tests for the semantic model by traversing the generated automaton, we can stimulate the implementation with the same inputs. This will drive the implementation model through the same states as that of the semantic model, except that there may be multiple refined transitions in the implementation model for one transition in the semantic. This makes it possible for us to use the exploration method for generating input sequences for corner case scenarios and execute them together with the implementation model.

The limitations of this approach have been detailed in Chapter 7. Some of them are that we

do not have a direct translation between the semantic model and the implementation. This requires implementation of both the models and may incur more cost in the validation effort than simply writing testbenches for the implementation model. Our validation approach also addresses a very specific part of the entire validation problem that is based on first ensuring a basic level of consistency between the two abstraction models by simulating testbenches. Furthermore, we do not extend this to all the heterogeneous MoCs though this can very well be done.

9.3 Dynamic Integration of Multiple Tools

The last question was "How can we enable dynamic integration of multiple third-party tools so that each tool acts as a service interacting with other services or clients through a common communicating meduim, and that these services can be dynamically turned on or off?" Here, we proposed the notion of service-orientation through middleware technology [123]. The middleware technology we propose is an OMG [64] standard with a variety of implementations for a variety of operating systems. The idea is to make the different third-party tools into services that can communicate with an object request broker (ORB). This needs a clear specification of the interface between the service and the ORB. This interface can then be used by any other services or even clients that interact with the ORB. This allows engineers to take advantage of multiple different tools without having to alter them for interoperation. Furthermore, these are dynamic services, so any unneeded ones may be turned off and only the ones required can be used. We even mention a use of this in integrating SpecExplorer with SystemC on all platforms in Chapter 7.

In summary, the questions we posed focused on the modeling, simulation, validation and dynamic integration of multiple tools aspects of an ESL methodology. Our solutions present proof of concepts for heterogeneous modeling, behavioral hierarchy, heterogeneous behavioral hierarchy, model-driven validation approach for directed test case generation and architecting a service-oriented environment for dynamically integrating multiple tools. These are all solutions for designs at the ESL.

Chapter 10

Conclusion and Future Work

Summary

This dissertation presents four ingredients that we feel are important for an ESL methodology to support. The ingredients we propose are heterogeneity, behavioral hierarchy, model-driven validation and a method for dynamically integrating and dissociating run-time validation environments. The objective of providing support for these ingredients is to empower designers with tools and methodologies for better design productivity by facilitating system level design. The acceptance of these in conferences [124, 125, 98, 126, 29, 103, 97, 127, 122] and journals [33, 121, 123] leads us to hope that these ingredients and their enabling technologies we proposed are slowly gaining acceptance.

Our previous work in extending SystemC with the capabilities of heterogeneous modeling and simulation demonstrated the increase in modeling fidelity [33, 32, 31]. Then, we extended SystemC's discrete-event simulation semantics with the synchronous dataflow (SDF), finite state machine (FSM) and communicating sequential processes (CSP) models of computation (MoC)s. We also performed simulation experiments in [33, 32, 31] that showed significant improvement in simulation performance for heterogeneous examples over the same examples designed using standard SystemC. This is particularly true for the SDF MoC (approximately 60% improvement) because an SDF design can be statically scheduled. Our extensions for the FSM and CSP MoCs are detailed in [32]. The FSM MoC did not show any improvement nor any degradation in performance, which nevertheless in our opinion is an advantage given that a higher level of modeling fidelity was achieved. This was improved by providing coding guidelines through which FSMs are described. The CSP extension showed simulation performance degradation by approximately 10%. This is due to the the manner in which the CSP scheduling was implemented and we believe that if the core kernel is optimized we can at least match the performance obtained by modeling CSP in standard SystemC.

In this dissertation, using our multi-MoC extension methodology, we extend SystemC with Bluespec's concurrent rule-based MoC that promotes a novel approach for hardware mod-

eling and simulation. We worked with Bluespec Inc. in developing this extension that is currently a commercial product [6]. However, since this is a first attempt at incorporating Bluespec's rule-based MoC with SystemC, we formally studied its interoperability with standard SystemC at the RTL abstraction layer. In doing this, we formally explain a gap in the simulation semantics and proposed a wrapper-based solution to bridge this gap.

Our second ingredient of behavioral hierarchy starts with a hierarchical FSM MoC, which we extend with a hierarchical SDF MoC to enable our multiple MoC extension support for heterogeneous behavioral hierarchy. The issue of adding multiple MoC support for heterogeneous behavioral hierarchy is non-trivial and requires defining and understanding semantics across MoCs. We describe an abstract semantics that enable heterogeneous behavioral hierarchy and a few details on our implementation to provide a better understanding of the interoperation. We present a hierarchical FSM example of an abstract power state model and a heterogeneous behavioral hierarchy example of the polygon infill processor. The former example neither shows significant increase in simulation speed nor does it show any degradation. The heterogeneous behavioral hierarchy example shows approximately 50% improvement. This in-fill processor example shows significant increase in simulation performance due to the preservation of behavioral hierarchy during simulation. This contrasts with the reference implementation of SystemC simulation kernel with structural hierarchy [97]. The reference implementation flattens all hierarchy during simulation.

Our experience in using extensions that support heterogeneous behavioral hierarchy has been positive. We felt that describing designs based on their inherent behaviors and having the support from the modeling and simulation framework to describe them made the modeling closer to designers intended computation. Thus, improving modeling fidelity. There are obvious advantages in simulation for specific MoCs, especially ones that can avoid the dynamic scheduling of SystemC. Aside from simulation, we feel that it is possible to synthesize hardware from heterogeneous descriptions because of the precise semantics of the MoCs and their interoperation. However, in our approach, we provide kernel-extensions and do not build our solution on top of standard SystemC. Since SystemC is an IEEE standard now [7], any extensions at the kernel-level is a departure from the standard. In that respect, HetSC [128] is more marketable for heterogeneity in SystemC. This is because they essentially create channels that generate events for the appropriate MoCs. This is actually mapping the heterogeneous MoCs to the discrete-event MoC of SystemC and this is what we exactly wish to not do in our work. This is because the underlying semantics used for computation is still discrete-event and not inherently the intended computation. Furthermore, it is very likely that simulation performance will suffer because of this mapping. Anyhow, we have not had the opportunity to do any simulation performance comparisons with their work as yet. This is because they are scheduled to release their source hopefully soon. So, this is something to do as future work.

The model-driven validation of SystemC designs is our third ingredient. This methodology allows designers to perform validation at essentially three levels. The first is at the semantic level (using SpecExplorer and AsmL) and the second at the implementation (SystemC). The

third is when the tests from SpecExplorer are executed in the implementation. An immediate benefit of this approach is that the semantic model is a formal and executable specification written in a language based on Abstract State Machines (ASM)s called AsmL. We formalized the discrete-event semantics in AsmL that allows designers to create semantic models in a similar fashion to SystemC. These are also simulated based on the discrete-event semantics. Because of the ASM formalism, it is possible to take this specification to model checkers for additional formal verification, but we do not explore this in this dissertation. Instead, we exploit the fact that SpecExplorer can generate automata from the AsmL specification. We use these to drive the implementation in SystemC. The challenge in this approach is primarily at the semantic exploration level. Exploration refers to using the AsmL description to generate an execution automaton. This requires good understanding of how the finite automaton is generated. There are various tricks that have to be used in generating the desired automaton and sometimes that involves altering the specification. Unfortunately, certain aspects of the exploration are not well documented. The only source are the research papers that describe the algorithms for generating the automata [110]. Another challenge we encountered was the fact that SystemC's implementation is in unmanaged C++, whereas SpecExplorer only allows for bindings to C# or Visual Basic. For this reason, we had to compile SystemC as a shared dynamic library and export certain important functions such as `sc_start` and `sc_stop`. We have to do the same for functions that drive the implementation model in SystemC. This has to then be imported into a C# class library, which is then loaded into SpecExplorer.

The last ingredient we propose in this dissertation is the dynamic integration and dissociation of integrating multiple runtime environments. We present CARH [123] that uses middleware technology to allow clients and services to interact with each other. As examples of services, we implemented a reflection and introspection, a logger, and an automated testbench generation services. We find that using middleware for gluing different tools is an extensible and tidy method. However, we experienced that our proof of concept suffered a simulation performance degradation due to the middleware. While such performance is often unacceptable for large models, it is possible to reduce this degradation significantly by customizing the CORBA implementation for the particular use of integrating tools. We simply used the one that was available to us instead of customizing one for this purpose. The advantage of such a validation environment software architecture is that any service or client that adheres to the OMG interfacing standard would be able to easily integrate. There are CORBA implementations for a variety of operating systems as well. This is another advantage because tools that are developed solely for Microsoft operating systems can easily communicate to the ones for Unix flavored operating systems and so on.

Future work

There are several opportunities for improving on the work presented in this dissertation. We feel that a full path from the specification, formal verification, modeling and simulation to

the implementation in hardware is crucial for making an ESL methodology marketable. A criticism that many ESL methodologies receive is that there is a lack of low-level information (power, area, thermal, timing, etc.) available at the higher abstraction layers to make the correct design decisions for the functionality and final implementation. So, it is important to focus on bringing these implementation details up to the system level abstraction layer as well.

The first improvement that could be done is to provide a specification framework for heterogeneous designs using ASMs. We know that a natural language description of a design is a poor abstraction for specification. This is one of the reasons we first implement our understanding of the specification in a language such as C/C++ that sometimes ends up serving as a reference model. However, we feel that languages such as C/C++ are themselves too low in abstraction to suffice as a good specification language. There are other alternatives such as capturing the specifications using the unified modeling language [129] (UML) and Rosetta [130, 131, 132]. The important aspect of all these specification languages is that they are executable.

In this dissertation, we used ASMs to capture the discrete-event semantics that allowed us to then capture the semantics of the intended designs using ASMs. The ASM formalism, in our opinion, is an appropriate formalism for hardware designers because ASMs are essentially fine granularity parallel transitions [79, 78]. This means that designers untrained in formal methods can easily adopt ASMs. So, as future work, this semantics library we present in this dissertation can be extended to incorporate other MoCs. This provides designers with a formal framework for specification and execution of heterogeneous designs. The additional advantage of using ASMs is that Microsoft SpecExplorer provides facilities for the execution and exploration of ASM specifications; thus leveraging existing research done at Microsoft Research. In addition, work by the authors of [45] claim to have used ASMs as an intermediate language for the formal verification of SystemC designs. The idea presented is impressive where the initial specification of the intended design and properties for verification are described in UML. This UML specification is then translated into ASMs. This is possible because both UML and ASMs have formal semantics. These are executable so that the properties can be validated on the intended design via simulation. Then, there is another translator to convert ASMs to SystemC. Similarly, the authors propose a method of extracting state machines from SystemC descriptions to generate ASMs for bringing the SystemC design to the ASM specification layer. Our work showed model-driven validation of SystemC designs. So, extending our methodology with additional MoCs at the semantic level and the verification flow, gives us a validation and verification flow for heterogeneous designs. We can also build the necessary translation tools to generate executable implementation in SystemC with the heterogeneous MoCs. Unfortunately, none of the tools mentioned by the authors of [45] are available for us to use and study.

For generating implementation code from our heterogeneous designs using the extensions for SystemC we need to have an intermediary compiler. For the Bluespec's rule-based MoC there is a compiler that generates synthesizable Verilog code and the same can be done

for the other MoCs. Since our MoCs have formal semantics, and their interoperability can also be formally described, it is possible to write compilers that correctly translates the heterogeneous designs into synthesizable Verilog. These are only some of the ideas that we have for the immediate future work and there may be others such as optimizing the implementation of the heterogeneous MoC extensions and developing a light-weight CORBA implementation for the integration of tools.

Bibliography

[1] G. Moore, "Cramming more Components onto Integrated Circuits," vol. 38, no. 8, April 1965.

[2] International Technology Roadmap for Semiconductors, "International Technology Roadmap for Semiconductors," http://public.itrs.net/, 2004.

[3] A. Sangiovanni-Vincentelli, "Reasoning About the Trends and Challenges of System Level Design," in *Proceedings of IEEE (to appear)*, 2007.

[4] B. Bailey, G. Martin, and A. Piziali, *ESL Design and Verification*. Morgan Kaufmann, 2007.

[5] Ptolemy Group, "Ptolemy II Website," http://ptolemy.eecs.berkeley.edu/ptolemyII/.

[6] Bluespec, "Bluespec," Website: http://bluespec.com/.

[7] IEEE, "SystemC IEEE 1666 Standard," Website: http://standards.ieee.org/getieee/1666/index.html.

[8] OSCI, "SystemC," Website: http://www.systemc.org.

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

[10] I. Sander and A. Jantsch, "System Modeling and Transformational Design Refinement in ForSyDe," in *Proceedings of IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 1, January 2004, pp. 17–32.

[11] SML-Sys Framework, "SML-Sys Website," http://fermat.ece.vt.edu/SMLFramework, 2004.

[12] SystemVerilog, "System Verilog," Website: http://www.systemverilog.org/.

[13] VHDL, "VHDL," Website: http://www.vhdl.org/.

[14] VERILOG, "Verilog," Website: http://www.verilog.com/.

[15] T. Grotker, S. Liao, G. Martin, and S. Swan, *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[16] R. A. Bergamaschi, "The Development of a High-Level Synthesis System for Concurrent VLSI Systems," Ph.D. dissertation, University of Southampton, 1989.

[17] F. Bruschi, F. Ferrandi, and D. Sciuto, "A Framework for the Functional Verification of SystemC Models," *International Journal of Parallel Programming*, vol. 33, no. 6, pp. 667–695, 2005.

[18] D. Kroening and N. Sharygina, "Formal Verification of SystemC by Automatic Hardware/Software Partitioning," in *Formal Methods and Models for Codesign*, 2005, pp. 101–110.

[19] P. Mishra, N. Dutt, N. Krishnamurthy, and M. Abadir, "A Top-Down Methodology for Validation of Microprocessors," in *IEEE Design & Test of Computers*, vol. 21, no. 2, 2004, pp. 122–131.

[20] P. Mishra, N. Krishnamurthy, N. Dutt, and M. Abadir, "A Property Checking Approach to Microprocessor Verification using Symbolic Simulation," in *Microprocessor Test and Verification (MTV), Austin, Texas, June*, 2002.

[21] Novas, "Debussy Debug System," http://www.novas.com.

[22] CoWare, "ConvergenSC," http://www.coware.com.

[23] Synopsys, "Smart RTL Verification," http://www.synopsys.com.

[24] Cadence, "Incisive Functional Verification," http://www.cadence.com.

[25] 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 *the proceedings of Design Automation Conference*, 2000, pp. 402–405.

[26] FERMAT, "SystemC-H," Website: http://fermat.ece.vt.edu/systemc-h/.

[27] S. Microsystems, "Java," http://java.com/.

[28] G. Kahn and D. MacQueen, "Coroutines and networks of parallel processes, Information Processing," in *Proceedings of IFIP Congress*, vol. 77, 1977, pp. 993–998.

[29] H. D. Patel and S. K. Shukla, "Towards Behavioral Hierarchy Extensions for SystemC," in *Proceedings of Forum on Design and Specification Languages*, 2005.

[30] F. Ghenassia, *Transaction-Level Modeling with SystemC*. Springer, 2005.

[31] H. D. Patel, "HEMLOCK: HEterogeneous ModeL Of Computation Kernel for SystemC," Master's thesis, Virginia Polytechnic Institute and State University, December 2003, website: http://scholar.lib.vt.edu/theses/available/etd-12052003-151950/-unrestricted/thesis_etd2.pdf.

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

[33] ——, "Towards a Heterogeneous Simulation Kernel for System Level Models: A SystemC Kernel for Synchronous Data Flow Models," in *IEEE Transactions in Computer-Aided Design*, vol. 24, August 2005, pp. 1261–1271.

[34] F. Herrera, P. Sánchez, and E. Villar, "Heterogeneous System-Level Specification in SystemC," in *Proceedings of Forum on Design and Specification Languages*, 2004.

[35] F. Herrera and E. Villar, "Mixing synchronous reactive and untimed models of computation in SystemC," in *Proceedings of Forum of Specification and Design Languages*, vol. 5, 2005.

[36] F. Herrera, P. Sánchez, and E. Villar, *Modeling and design of CSP, KPN and SR systems in SystemC*. Kluwer Academic Publisher, 2004, pp. 133–148.

[37] S. Thompson, "Haskell - The Craft of Functional Programming, 2 ed," in *MA: Addison-Wesley*, 1999.

[38] A. Jantsch, *Modeling Embedded Systems And SOC's - Concurrency and Time in Models of Computations*. Morgan Kaufmann Publishers, 2001.

[39] D. Rosenband and Arvind, "Modular Scheduling of Guarded Atomic Actions," in *Proceedings of the Design Automation Conference*, June 2004, pp. 55–60.

[40] N. Dave, M. C. Ng, and Arvind, "Automatic Synthesis of Cache-Coherence Protocol Processors Using Bluespec," in *Proceedings of Formal Methods and Models for Codesign*, July 2005, pp. 25–34.

[41] J. Hoe and M. Arvind, "Hardware Synthesis from Term Rewriting Systems," in *Proceedings of the IFIP TC10/WG10. 5 Tenth International Conference on Very Large Scale Integration: Systems on a Chip*. Kluwer, BV Deventer, The Netherlands, The Netherlands, 1999, pp. 595–619.

[42] F. Baader and T. Nipkow, *Term Rewriting and All That*. Cambridge University Press, 1998.

[43] J. C. Hoe and Arvind, "Synthesis of operation-centric hardware descriptions," *IEEE International Conference on Computer Aided Design*, pp. 511–518, 2000.

[44] A. Gawanmeh, A. Habibi, and S. Tahar, "Enabling SystemC Verification using Abstract State Machines," Department of Electrical and Computer Engineering, Concordia University, Tech. Rep., 2004.

[45] A. Habibi and S. Tahar, "Design and Verification of SystemC Transaction-Level Models," in *IEEE Transactions on Very Large Scale Integration Systems*, vol. 14, no. 1, 2006, pp. 57–68.

[46] A. Habibi, H. Moinudeen, and S. Tahar, "Generating Finite State Machines from SystemC," in *Proceedings of Design, Automation and Test in Europe*, 2006, pp. 76–81.

[47] W. Mueller, J. Ruf, D. Hoffmann, J. Gerlach, T. Kropf, and W. Rosenstiehl, "The Simulation Semantics of SystemC," in *Proceedings of Design, Automation and Test in Europe*, 2001, pp. 64–70.

[48] W. Snyder, "SystemPerl," http://www.veripool.com/systemperl.html.

[49] Edison Design Group C++ Front-End, "Edison Group Front-End," Website: http://edg.com/cpp.html.

[50] F. Doucet, S. Shukla, and R. Gupta, "Introspection in System-Level Language Frameworks: Meta-level vs. Integrated," in *Proceedings of Design and Test Automation in Europe*, 2003, pp. 382–387.

[51] GreenSocs, "Pinapa: A SystemC Frontend," http://www.greensocs.com/.

[52] Doxygen, "Doxygen," http://www.stack.nl/ dimitri/doxygen/.

[53] The Apache XML Project, "Xerces C++ Parser," http://xml.apache.org/xerces-c/.

[54] Forschungszentrum Informatik, "KaSCPar - Karlsruhe SystemC Parser Suite," http://www.fzi.de/sim/kascpar.html.

[55] S. Microsystems, "Java," https://javacc.dev.java.net/.

[56] J. Lapalme, E. M. Aboulhamid, G. Nicolescu, L. Charest, F. R. Boyer, J. P. David, and G. Bois, ".NET Framework–A Solution for the Next Generation Tools for System-Level Modeling and Simulation," vol. 1. IEEE Computer Society Washington, DC, USA, 2004.

[57] Mono, "Mono Project," http://www.mono-project.com/.

[58] L. Charset, M. Reid, E. M. Aboulhamid, and G. Bois, "A methodology for interfacing open source SystemC with a thirdparty software," in *Proceedings of Design and Test Automation in Europe*, 2001, pp. 16–20.

[59] OMG, "OMG CORBA," http://www.corba.org/.

[60] A. Ledeczi, M. Maroti, A. Bakay, and G. Karsai, "The Generic Modeling Environment," Vanderbilt University. Institute for Software Integrated Systems Nashville, 2001.

[61] ACE, "Adaptive Communication Environment," http://www.cs.wustl.edu/ schmidt/ACE.html.

[62] E. Gamma, R. Helm, J. R, and J. Vlissides, *Design Patterns*. Addison Wesley, 1995.

[63] TAO, "Real-time CORBA with TAO (The ACE ORB)," http://www.cs.wustl.edu/ schmidt/TAO.html.

[64] OMG, "OMG," http://www.omg.org/.

[65] A. Amar, P. Boulet, J. Dekeyser, S. Meftali, S. Niar, M. Samyn, and J. Vennin., "SoC Simulation," http://www.inria.fr/rapportsactivite/RA2003/dart2003/.

[66] L. Formaggio and G. P. F. Fummi, "A Timing-Accurate HW/SW Co-simulation of an ISS with SystemC," in *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, 2004, pp. 152–157.

[67] N. S. 2, "Network Simulator 2," Website: http://www.isi.edu/nsnam/ns/.

[68] Hamabe, "SystemC over CORBA," http://www5a.biglobe.ne.jp/ hamabe/index.html.

[69] D. A. Mathaikutty, H. D. Patel, and S. K. Shukla, "A Multi-MOC Framework for SOC Modeling using SML," Virginia Tech, Tech. Rep. 2004-04, 2004.

[70] E. A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing," in *Proceedings of IEEE Transactions on Computers*, ser. 1, vol. C-36, 1987.

[71] S. Bhattacharyya, P. Murthy, and E. Lee, *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, 1996.

[72] M. Chiodo and P. Giusto and H. Hsieh and A. Jurecska and L. Lavagno and A. Sangiovanni-Vincentelli, "Hardware-software codesign of embedded systems," in *Proceedings of IEEE Micro*, 1994, pp. 26–36.

[73] M. Pankert and O. Mauss and S. Ritz and H. Meyr, "Dynamic data flow and control flow in high level DSP code synthesis," in *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing*, vol. 2, 1994, pp. 449–452.

[74] D. Harel, "Statecharts: A visual formalism for complex systems," in *Scientific Computing Program*, vol. 8, 1987, pp. 231–274.

[75] A. Girault and B. Lee and and E. A. Lee, "Hierarchical Finite State Machines with Multiple Concurrency Models," in *Proceedings of IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, 1999, pp. 742–760.

[76] C. Hoare, "Communicating Sequential Processes," in *Communications of the ACM*, ser. 8, vol. 21, 1978.

[77] C. A. R. Hoare, *Communicating Sequential Processes*. Prentice Hall, 1985.

[78] E. Börger and R. Stärk, *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer, 2003.

[79] Y. Gurevich, *Evolving algebras 1993: Lipari Guide*. Oxford University Press, 1995, pp. 9–36, in Specification and Validation Methods.

[80] SpecExplorer, http://research.microsoft.com/specexplorer/.

[81] M. Barnett, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes, "Towards a tool environment for model-based testing with AsmL," *Petrenko and Ulrich, editors, Formal Approaches to Software Testing, FATES*, vol. 2931, pp. 264–280, 2003.

[82] Y. Gurevich, B. Rossman, and W. Schulte, "Semantic essence of AsmL," *Theoretical Computer Science*, vol. 343, no. 3, pp. 370–412, 2005.

[83] M. Barnett and W. Schulte, "The ABCs of specification: AsmL, behavior, and components," *Informatica*, vol. 25, no. 4, pp. 517–526, 2001.

[84] Microsoft Research, "Spec#," Website:http://research.microsoft.com/specsharp/.

[85] M. Barnett, K. Leino, and W. Schulte, "The Spec# programming system: An overview," *CASSIS 2004: International Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices*, 2004.

[86] Formal Engineering Research with Models, Abstractions and Transformations, "Formal Engineering Research with Models, Abstractions and Transformations," Website: http://fermat.ece.vt.edu/.

[87] C. Campbell, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillman, and M. Veanes, "Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer," Microsoft Research, Tech. Rep. MSR-TR-2005-59, 2005.

[88] Boost, "Boost graph library," Website: http://www.boost.org.

[89] J. Bresenham, "Bresenham's Line Drawing Algorithm," http://www.cs.helsinki.fi/group/goa/mallinnus/lines/bresenh.html.

[90] FORTE, "Forte Design Systems," Website: http://www.forteds.com/.

[91] Celoxica, "Celoxica's Agility Compiler," Website: http://www.celoxica.com/products/agility/.

[92] Tensilica, "XPRES Compiler," Website: http://www.tensilica.com/products/xpres.htm.

[93] Mentor Graphics, "Catapult C Synthesis," Website:http://www.mentor.com/.

[94] E. A. Lee, "The problem with threads," *Computer*, vol. 39, no. 5, pp. 33–42, 2006.

[95] E. Team, "SIGNAL: A polychronous data-flow language," Website: http://www.irisa.fr/espresso/Polychrony/.

[96] Esterel-Technologies, "Lustre," Website: http://www-verimag.imag.fr/ synchron/.

[97] H. D. Patel and S. K. Shukla, "Heterogeneous Behavioral Hierarchy for System Level Designs," in *Proceedings of Design Automation and Test in Europe*, 2006.

[98] ——, "Deep vs. Shallow, Kernel vs. Language - What is Better for Heterogeneous Modeling in SystemC?" in *Proceedings of Microprocessor Test and Verification*, 2006.

[99] M. Pellauer, M. Lis, D. Baltus, and R. Nikhil, "Synthesis of Synchronous Assertions with Guarded Atomic Actions," in *Formal Methods and Models for Codesign*, 2005.

[100] GreenSocs, "Enabling ESL Interoperability," http://www.greensocs.com/.

[101] D. Rosenband and Arvind, "Hardware synthesis from guarded atomic actions with performance specifications," in *Proceedings of International Conference on Computer Aided Design*, November 2005, pp. 784–791.

[102] H. D. Patel and S. K. Shukla, "Model-driven validation of SystemC designs," FERMAT Lab. Virginia Tech., Tech. Rep. 2006-18, 2006.

[103] H. D. Patel, S. K. Shukla, E. Mednick, and R. Nikhil, "A Rule-based Model of Computation for SystemC: Integrating SystemC and Bluespec for Co-Design," in *Proceedings of Formal Methods and Models for Codesign*, 2006, pp. 39– 48.

[104] M. D. Riedel and J. Bruck, "The Synthesis of Cyclic Combinational Circuits," in *Proceedings of Design Automation Conference*, 2003, pp. 163 – 168.

[105] S. A. Edwards, "Making Cyclic Circuits Acyclic," in *Proceedings of Design Automation Conference*, 2003, pp. 159–162.

[106] G. Berry, *The Constructive Semantics of Pure Esterel*. Draft book, available at http://esterel.org, version 3, 1999.

[107] L. Stok, "False loops through resource sharing." IEEE Computer Society Press Los Alamitos, CA, USA, 1992, pp. 345–348.

[108] A. Yu, C. Hsut, and M. Lee, "Eliminating False Loops Caused by Sharing in Control Path," *Proceedings of the 9th International Symposium on System Synthesis*, vol. 1080, p. 96, 1820.

[109] E. A. Lee and A. L. Sangiovanni-Vincentelli, "Comparing Models of Computation," in *In Proceedings of IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 17, no. 12. IEEE Computer Society, 1998, pp. 1217–1229.

[110] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes, "Generating Finite State Machines from Abstract State Machines," in *Proceedings of the international symposium on Software testing and analysis*. ACM Press New York, NY, USA, 2002, pp. 112–122.

[111] S. A. Sharad and S. K. Shukla, *Formal Methods And Models For System Design: A System Level Perspective*. Kluwer Academic Publisher, 2005, pp. 317–331.

[112] Borland, "Borland VisiBroker: A Robust CORBA Environment for Distributed Processing," Website: http://www.borland.com/us/products/visibroker/index.html.

[113] XASM Group, "XASM: eXtensible Abstract State Machines," Website:http://www.xasm.org/.

[114] CoreASM Group, "CoreASM: An Extensible ASM Execution Engine," Website:http://www.coreasm.org/.

[115] SPECC, "SpecC," Website: http://www.ics.uci.edu/specc/.

[116] Synopsys, "Smart RTL Verification," http://www.synopsys.com.

[117] D. Berner, H. Patel, D. Mathaikutty, S. Shukla, and J. Talpin, "SystemCXML: An Extensible SystemC Front End Using XML," in *Prcoeedings of Forum on Design and Specification Languages*, 2005.

[118] E. R. Gansner and S. C. North, "An open graph visualization system and its applications to software engineering," *Softw. Pract. Exper.*, vol. 30, no. 11, pp. 1203–1233, 2000.

[119] D. A. Mathaikutty, D. Berner, H. D. Patel, and S. K. Shukla, "FERMAT's SystemC Parser," http://systemcxml.sourceforge.net, 2004.

[120] Troll Tech, "Qt," Website: http://troll.no.

[121] H. D. Patel and S. K. Shukla, "Heterogeneous Behavioral Hierarchy for System Level Designs," in *Proceedings of IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, 2007, pp. 565–570.

[122] ——, "Model-driven validation of SystemC designs," in *Proceedings of Design Automation Conference (to appear)*, 2007.

[123] D. A. Mathaikutty, H. D. Patel, and S. K. Shukla, "CARH: A Service Oriented Architecture for Dynamic Verification of SystemC Models," in *Proceedings of IEEE Transactions on Computer Aided Design*, vol. 25, no. 8, 2006, pp. 1458–1474.

[124] 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 *Proceedings of International Symposium in VLSI.* IEEE Computer Society Press, 2004, pp. 241–242.

[125] ——, "Towards A Heterogeneous Simulation Kernel for System Level Models: A SystemC Kernel for Synchronous Data Flow Models," in *Proceedings of ACM Great Lakes Symposium in VLSI*, 2004, pp. 248–253.

[126] D. Berner, D. A. Mathaikutty, H. D. Patel, and S. K. Shukla, "Automated Extraction of Structural Information from SystemC-based IP for Validation," in *Proceedings of Microprocessor Test and Verification*, 2005, pp. 99–104.

[127] H. D. Patel and S. K. Shukla, "Tackling an Abstraction Gap: Co-simulating SystemC DE with Bluespec ESL," in *Proceedings of Design Automation and Test in Europe (to appear)*, 2007.

[128] F. Herrera and E. Villar, "A framework for embedded system specification under different models of computation in SystemC," in *Proceedings of Design Automation Conference*, 2006, pp. 911–914.

[129] OMG Group, "Unified Modeling Language," Website:http://www.uml.org/.

[130] C. Kong and P. Alexander, "Modeling model of computation ontologies in Rosetta," in *Proceedings of the Formal Specification of Computer-Based Systems Workshop (FSCBS), Lund, Sweden, March*, 2002.

[131] ——, "The Rosetta meta-model framework," in *Proceedings. 10th IEEE International Conference and Workshop on the*, 2003, pp. 133–140.

[132] C. Kong, P. Alexander, and C. Menon, "Defining a Formal Coalgebraic Semantics for The Rosetta Specification Language," in *Journal of Universal Computer Science*, vol. 9, no. 11. Institute for Information Processing and Computer Supported New Media, Graz University of Technology, 2003, pp. 1322–1349.

[133] H. D. Patel, S. Gupta, S. K. Shukla, and R. Gupta, "Design Issues in Embedded Systems," In 'Embedded Systems', CRC Press, 2004.

[134] ——, "A survey of networked embedded systems: An introduction," In the Handbook of Information Technology, CRC Press, 2004.

[135] P. Kachroo, S. Shukla, T. Erbes, and H. D. Patel, "Stochastic Learning Feedback Hybrid Automata for Power Management in Embedded Systems," in *Proceedings of the IEEE Workshop on Soft Computing in Industrial Applications (SMCia'03)*, 2003, pp. 121–125.

[136] J. Armstrong, S. Agrawal, H. D. Patel, and S. K. Shukla, "PROGRESS: PROcess GRaph for Embedded Systems and Software: Combining top-down and bottom-up System Design Methodology," in *Proceedings of the Electronic Design Processes Workshop*, 2003.

[137] D. A. Mathaikutty, H. D. Patel, and S. K. Shukla, "EWD: A Metamodeling Driven Customizable Multi-MoC System Modeling Environment," in *Proceedings of ACM Transactions on Design Automation of Electronic Systems (to appear)*, 2007.

[138] S. Sharad, D. Bhaduri, M. Chandra, H. Patel, and S. Syed, "Systematic Abstraction of Microprocessor RTL models to enhance Simulation Efficiency," in *Proceedings of Microprocessor Test and Verification*, 2003, pp. 103–108.

[139] D. A. Mathaikutty, H. D. Patel, and S. K. Shukla, "UMoC++: Modeling Environment for Heterogeneous Systems based on Generic MoCs," in *Proceedings of Forum of Specification and Design Languages*, 2005.

[140] ——, "A Functional Programming Framework of Heterogeneous Model of Computations for System Design," in *Proceedings of Forum of Specification and Design Languages*, 2004.

Vita

Hiren Dhanji Patel

Personal Data

Date of Birth	:	January 31, 1980
Country of Birth	:	Dar-es-Salaam, Tanzania
Nationality	:	Tanzanian
Marital Status	:	Single
Visa Status	:	F-1

Office Address:

FERMAT Research Lab.
302 Whittemore Hall
Blacksburg, VA 24060
Email: hiren@vt.edu
URL: http://fermat.ece.vt.edu/Fermatian_Info/hiren.html

Internship: No

Full Time Employment: Yes
Availability Date: May 2007
University: Virginia Polytechnic Institute and State University
Faculty Advisor: Sandeep K. Shukla
Citizenship: Tanzanian

Objective

Seeking a research and development position in the electronic design automation and design industry; software and hardware design and development industry.

Research Interests

(a) Embedded Systems Design

(b) Hardware & Software Co-Design Methodologies

(c) System Level Design and Validation

(d) System Level Design Frameworks, Tools and Methodologies

(e) Models of Computation

(f) Software Design

(g) Distributed Software Services

Education

(a) Ph.D., Computer Engineering, (Expected May 2007), Virginia Polytechnic Institute and State University.

Ph.D. Dissertation Title: "**ISLAND**: Ingredients for Successful System Level Automation & Design Methodology –*Support for Multiple Models of Computation, Heterogeneous Behavioral Hierarchy, Reflection & Introspection, and Service-Oriented tool Integration Environment*

(b) M.S., Computer Engineering, (December 2003), Virginia Polytechnic Institute and State University.

M.S. Thesis Title: "HEMLOCK: HEterogeneous ModeL Of Computation Kernel for SystemC"
URL: `http://www.fermat.ece.vt.edu/hiren/thesis.pdf`

(c) B.S., Computer Engineering, December 2001, Virginia Polytechnic Institute and State University.

Current Work

(a) Formalization: Studying and providing formal semantics for heterogeneous behavioral hierarchy and Models of Computation.
Providing formal semantics for heterogeneous behavioral hierarchy with respect to SystemC discrete-event MoC with hierarchical FSM and SDF MoCs. In addition,

the semantics associated with employing Bluespec's rule-based MoC within SystemC. Investigating the use of Abstract State Machines (ASM) and variants of ASMs for formal specification followed with an executable specification in Microsoft Corporation's AsmL.

(b) Directed test case generation: Test case generation using SpecExplorer and Abstract State Machines.

Employ SpecExplorer and its semantic modeling and simulation capabilities of Abstract State Machines to provide a methodology for semantic modeling for Discrete-Event models. Employ tools from SpecExplorer for exploring state space in semantic model and generate directed test cases. Exercise these test cases on the implementation model in SystemC via a set of wrappers.

(c) SystemC Parsing: Extracting SystemC structural and behavioral information using EDG (Student advising)

Static analysis of SystemC source code using commercial C/C++ front-end parser from the Edison Front-end Group (EDG). Parse all SystemC structural and behavioral constructs and represent in an intermediary structure for further additional processing.

(d) Eclipse-based SystemC IDE: SystemC integrated development environment for modeling and simulation (Student advising)

An Eclipse-based SystemC modeling, development and simulation environment. Some of the supported features are managed make projects, syntax highlighting, code folding, content assist and outline manipulation.

(e) Bluespec: Introducing an action-oriented Model of Computation kernel extension for SystemC
Working with Bluespec in introducing an action-oriented Model of Computation extension for SystemC for hardware designs. Devise and implement constructs to represent an action-oriented Model of Computation and investigate semantics for interoperating with SystemC.

(f) Heterogeneous Behavioral Hierarchy Models of Computation Kernel Extensions for SystemC
Kernel development to introduce heterogeneity and hierarchy in SystemC by implementing a variety of Models of Computation in SystemC's simulation framework. Recent development has heterogeneous hierarchy enabled for the Finite State Machine (FSM) and Synchronous Data Flow (SDF) Models of Computation available at http://fermat.ece.vt.edu/tools/hierarchy. Investigate behavioral hierarchy across different Models of Computation, their significance, usage modes, and implementation for extensibility and integration with SystemC.

(g) Distributed Service Oriented Frameworks
Proposed an architecture based on Adaptive Communication Environment (ACE) and

The Ace ORB (TAO) way, to implement a service oriented architecture for validation and verification of SystemC models, CARH. Continue to investigate implementation of automated test generators, coverage monitors, performance services etc. CARH has the notion of reflection and introspection capabilities for SystemC models. CARH is available at
http://fermat.ece.vt.edu/tools/carh.

Skills

– C++, SystemC, Verilog, ACE, TAO, AsmL, CORBA, Ptolemy II, Qt, Winsock API, VHDL, OPNET, Intel/Motorola Assembly, Perl, Turbo Pascal.

– Linux and Microsoft Windows.

– Languages: English and Gujurati.

Experience

(a) January 2003 to Present: Graduate Research Assistant, Virginia Tech., Blacksburg, VA.

- Explore possibilities of a methodology for validation of SystemC designs using SpecExplorer. SpecExplorer's Spec# and AsmL language (based on ASM) is used to provide semantic models that are simulated with our discrete-event simulation semantics also specified in AsmL. The semantic model is used for exploration purposes and test case generation. Test cases can be directed for reachability of particular states of the system. Additional wrappers for SystemC and C# are added to allow SpecExplorer to execute the directed test cases in the semantic model alongside the implementation model in SystemC for conformance checking.
- Advise an undergraduate student in employing the Edison Group C/C++ front-end parser for extracting structural and behavioral information of SystemC designs. An internal representation to store design information opens up avenues for analysis, transformations and translations.
- Advise an undergraduate student using Eclipse to create a plug-in for SystemC modeling and simulation. Currently working with existing projects such as GreenSoCs Eclipse plugin to provide better features.
- Investigate the idea of behavioral hierarchy in system level design languages and frameworks by introducing heterogeneous hierarchical MoCs for SystemC for PhD dissertation.

- Investigating SystemC 2.0.1 simulation kernel source to implement different Models of Computation directly through simulation kernel programming. Successful incorporation of Discrete-Event and Synchronous Data Flow Models of Computation presented in M.S. Thesis.
- Implemented addition kernels for Finite State Machine and Communicating Sequential Processes Models of Computation such that all Models of Computation are interoperable and we detail our work in a book published by Kluwer.
- Investigate EWD, a metamodeling driven customizable multi-MoC framework for system level modeling. This framework utilizes a metamodeling framework called the Generic Modeling Environment (GME) and with a design flow, we can construct designs in a component-based visual environment and automatically generate executable code for Standard ML.
- Investigating CARH, a service oriented architecture for validation and verification. We employ a CORBA-variant called The ACE (Adaptive Communication Environment) Orb (TAO) to allow users to implement their own services for facilities such as model visualization, reflection and introspection, tracing, debugging and coverage monitors.
- Investigated EDG C++ Front-end parsing for SystemC. After some effort, we resorted to using Doxygen along with XML utilities to construct the FERMAT's SystemC parser.
- Compiler design using yacc and flex.

(b) May 2006 to August 2006: Research Engineer, IBM Research T. J. Watson, Yorktown Heights, NY.
Standardized approach for SystemC component composition.

- Investigate methods of incorporating OSCI transaction level modeling standards into their existing model infrastructure.
- Provide a methodology for integrating models at different abstractions, transferring varying transactions and some that require specializing communication.
- Innovated a refinement-based methodology for creating transactors/adaptors to bridge the gap between transaction boundaries and specialization of communication.
- Provided a generic request/response based communication mechanism within the adaptors for a variety of communication channels such as pipeline-based communication and STL queue-based.
- Used the generic adaptor mechanism for the power management unit infrastructure.

(c) August 2005 to May 2006: Research Engineer/Consultant, Bluespec Inc., Waltham, MA
Provide Bluespec system description and simulation environment in SystemC as extensions for SystemC.

- Introduce new language syntax to describe Bluespec rule-based designs as done in Bluespec SystemVerilog.
- Interface new language description with Bluespec-SystemC kernel with respective updates to the scheduling implementation.
- Product information available at http://bluespec.com. Free to the community version provided at http://bluespec.com/products/ESLSynthesisExtensions.htm.

(d) May 2005 to August 2005: Research Engineer, IBM Research T. J. Watson, Yorktown Heights, NY.
Investigate and implement a toolkit for micro-architectural exploration for SystemC.

- Take existing performance models and rework the modules to allow complete parameterization and modularization. This allows the modules to be plugged into other designs and configured with ease.
- Innovate a connection mechanism reducing the tedious addressing of port and signal bindings between the modules.
- Incorporate hooks into the models for performance measurement, area analysis and debugging.
- Introduce a tracing mechanism through the innovative connection mechanism to trace specific instructions, set of instructions, or all instructions processed in an architectural model.
- Construct a visual environment based on Tcl/Tk for representing the performance and area measurements.

(e) June 2004 to August 2004: Research Engineer, Calypto Design Systems, Santa Clara, CA.
Generating simulation metrics for SystemC versus Verilog models and analyzing performance hotspots in SystemC models. Aid in developing test suites, debugging of Calypto proprietary software, improvement opportunities in respect to performance.

- Investigate performance tradeoffs between Calypto proprietary SystemC and Verilog models and opportunities for increase in simulation efficiency.
- Co-simulate SystemC and Verilog models using VPI/PLI and analyze performance degradation hotspots.
- Propose better translation schemes from Verilog to SystemC.

(f) June 2003 to August 2003: Software Test Engineer, Microsoft Corporation, Redmond, WA.
Aiding in investigating tools that better the roles of testers providing opportunity for self-learning of combinatorial tools, programming paradigms and more importantly the problems test teams are concerned with regarding efficient test case generation, build server information deployment and automation of certain tasks.

- A C# WMI solution was delivered to retrieve server-side information through files, registries and any locally accessible medium to perform simultaneous server updates. Client and Server side both programmed into a WMI service provider into a static namespace providing event triggers for automatic refresh at initial connection from remote computer.
- Combinatorial tools were explored to provide the current testers information on using the combinatorial tool as well as the plans for integration of tool with User Interface automation framework.
- Performed tester role by writing test cases, running test passes, investigating and resolving bugs.

(g) August 2002 to December 2002: Graduate Grader, Virginia Tech., Blacksburg, VA.
Network programming using C++ and Winsock API and evaluating and assisting students with technical projects and implementation.

(h) June 2002 to August 2002: Graduate Teaching Assistant, Virginia Tech., Blacksburg, VA.
Assisting and grading students in Digital Design of systems using VHDL and Xilinx FPGA boards.

(i) May 2002 to June 2002: Network Administrator, TVWordlwide.com, Chantilly, VA.
Maintained, configured and deployed web, video, mail and database servers. Configured network switches, routers, firewalls, and RAID backup systems. Managed and administered Windows NT and Linux servers for network operations.

Journals

D. A. Mathaikutty, H. D. Patel, Sandeep K. Shukla and Axel Jantsch, *EWD: A Metamodeling Driven Customizable Multi-MoC System Modeling Environment*, to appear in proceedings of Transactions on Design Automation of Electronic Systems (TOADES), 2007.

H. D. Patel and S. K. Shukla, *Heterogeneous Behavioral Hierarchy Extensions for SystemC*, in proceedings of IEEE Transactions in Computer-Aided Design, pp. 765-780, April 2007.

H. D. Patel, D. A. Mathaikutty, D. Berner and Sandeep Shukla, *CARH: A Service Oriented Architecture for Validating System Level Designs*, in proceedings of IEEE Transactions in Computer-Aided Design,Volume 25, Issue 8, pp. 1458-1474, August 2006.

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 Transactions in Computer-Aided Design, Vol. 24, No. 8, pp. 1261-1271, August 2005.

Conference

H. D. Patel and S. K. Shukla, *Model-driven validation of SystemC designs*, to appear in proceedings of Design Automation Conference, 2007.

H. D. Patel and S. K. Shukla, *Tackling an Abstraction Gap: Co-simulating SystemC DE with Bluespec ESL*, to appear in proceedings of Design Automation and Test in Europe, 2007.

H. D. Patel and S. K. Shukla, *Deep vs. Shallow, Kernel vs. Language – What is Better for Heterogeneous Modeling in SystemC?*, to appear in Proceedings of Microprocessor Test and Verification, 2006.

H. D. Patel, S. K. Shukla, E. Mednick and R. S. Nikhil, *A Rule-based Model of Computation for SystemC: Integrating SystemC and Bluespec for Co-design*, in Formal Methods and Models for Codesign, pp. 39-48, Napa Valley, California, 2006.

H. D. Patel and S. K. Shukla, *Heterogeneous Behavioral Hierarchy for System Level Design*, in Proceedings Design Automation and Test in Europe, pp. 565-570, Munich, Germany, March 2006.

D. Berner, H. D. Patel, D. Mathaikutty and S. K. Shukla, *Automated Extraction of Structural Information from SystemC-based IP for Validation*, In Proceedings of Microprocessor Test and Verification (MTV '05), pp. 99-104, Austin, Texas 2005.

D. Berner, H D. Patel, D. A. Mathaikutty and S. K. Shukla, *SystemCXML: An Extensible SystemC Frontend Using XML*, in proceedings of Forum on Design and Specification Languages (FDL '05), Lausanne, Switzerland, September 2005.

D. A. Mathaikutty, H. D. Patel, S. K. Shukla and A. Jantsch, *UMoC++: Modeling Environment for Heterogeneous Systems based on Generic MoCs*, in proceedings of Forum on Design and Specification Languages (FDL '05), Lausanne, Switzerland, September 2005.

H. D. Patel and S. K. Shukla, *Towards Behavioral Hierarchy Extensions for SystemC*, in proceedings of Forum on Design and Specification Languages (FDL '05), Lausanne, Switzerland, September 2005.

D. A. Mathaikutty, H. D. Patel and S. K. Shukla, *A Functional Programming Framework of Heterogeneous Model of Computation for System Design*, in proceedings of Forum on Design Languages (FDL '04), Lille, France, September 2004.

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 proceedings of Great Lakes Symposium on VLSI (GLSVLSI '04), pp 248-253, Boston, Massachusetts, April 2004.

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 proceedings of International Symposium in VLSI, IEEE Computer Society Press, pp 240-242, February, Lafayette,

Louisiana, 2004.

S. Sharad, D. Bhaduri, M. Chandra, H. D. Patel, and S. Syed, *Systematic Abstraction of Microprocessor RTL models to enhance Simulation Efficiency*, in proceedings of Microprocessor Test and Verification (MTV '03), pp 103-108, Austin, Texas, September 2003.

P. Kachroo, S. Shukla, T. Erbes, and H. D. Patel, *Stochastic Learning Feedback Hybrid Automata for Power Management in Embedded Systems*, the proceedings of the IEEE Workshop on Soft Computing in Industrial Applications (SMCia'03), pp 121-125, Binghamton, New York, June 2003.

Books

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

Book Chapters

D. A. Mathaikutty, H. D. Patel, S. K. Shukla and A. Jantsch, *UMoC++: Modeling Environment for Heterogeneous Systems based on Generic MoCs*, in Advances in Design and Specification Languages for SoCs - Selected Contributions from FDL'05, Chapter 7. Springer Verlag, 2006.

H. D. Patel, S. K. Shukla. Edited by R. Gupta, P. Le Guernic, S. Shukla, and J. P. Talpin, *Truly Heterogeneous Modeling with SystemC*, ch. Formal Models and Methods for System Design, pp. 88-101, Kluwer Academic Publishers, The Netherlands, 2004.

H. D. Patel, S. Gupta, S. K. Shukla, and R. Gupta, *Design Issues in 'Embedded Systems'*, CRC Press, 2004.

H. D. Patel, S. Gupta, S. K. Shukla, and R. Gupta, *A survey of networked embedded systems: An introduction*, in the Handbook of Information Technology, CRC Press, 2004.

Selected Programming Projects

(a) Model-driven validation of SystemC using SpecExplorer

(b) Heterogeneous hierarchy for System Level Design Languages (SystemC).

(c) SystemCXML: FERMAT's SystemC Parser using Doxygen and XML.

(d) HEMLOCK: HEterogeneous ModeL Of Computation Kernel for SystemC.

(e) CARH: A Service-oriented Architecture for Validating System Level Designs

(f) FEDG: FERMAT's EDG-based SystemC Parser

(g) FSCIDE: FERMAT's Eclipse-based SystemC IDE

References

(1) Professor Sandeep K. Shukla
Bradley Department of Electrical and Computer Engineering,
Virginia Polytechnic Institute and State University
340 Whittemore Hall
Blacksburg, VA 24061
Tel: (540) 231-2133
Fax: (540) 231-3362
Email: `shukla@vt.edu`

(2) Reinaldo Bergamaschi
IBM Research T.J. Watson
1101 Kitchawan Rd.
Yorktown Heights, NY 10598
Tel: (914) 945-3903
Email: `berga@us.ibm.com`

(3) Sumit Roy
Calypto Design Systems,
2933 Bunker Hill Lane
Suite 202,
Santa Clara, CA 95054
Tel: (408) 850-2300
Fax: (408) 850-2301
Email: `sroy@calypto.com`