Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2008, Article ID 519474, 14 pages
doi:10.1155/2008/519474

Research Article

Model-Driven Validation of SystemC Designs

Hiren D. Patel[1] and Sandeep K. Shukla[2]

[1] *Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, 545K Cory Hall, Berkeley, CA 94720, USA*
[2] *Formal Engineering Research with Models, Abstractions and Transformations, Virginia Tech., 302 Whittemore Hall, Blacksburg, VA 24060, USA*

Correspondence should be addressed to Hiren D. Patel, hiren@eecs.berkeley.edu

Received 1 October 2007; Revised 24 February 2008; Accepted 7 April 2008

Recommended by Axel Jantsch

Functional test generation for dynamic validation of current system level designs is a challenging task. Manual test writing or automated random test generation techniques are often used for such validation practices. However, directing tests to particular reachable states of a SystemC model is often difficult, especially when these models are large and complex. In this work, we present a model-driven methodology for generating directed tests that take the SystemC model under validation to specific reachable states. This allows the validation to uncover very specific scenarios which lead to different corner cases. Our formal modeling is done entirely within the Microsoft SpecExplorer tool to formally describe the specification of the system under validation in the formal notation of AsmL. We also exploit SpecExplorer's abilities for state space exploration for our test generation, and its APIs for connecting the model to real programs to drive the validation of SystemC models with the generated test cases.

1. INTRODUCTION

SystemC [1] has gained considerable traction in the electronic design automation community as an entry language for system-level modeling, simulation, and validation. It is often used in the early stages of a product's design cycle for fast design space exploration. This means that an ample amount of time is spent in simulating and validating the system-level design. The standard procedure for validation requires a designer to generate a set of input sequences that exercise all the relevant and important properties of the system. Randomized test generation and simulation are almost always augmented with manual test sequences for properties that are critical to the functionality of the system. Manually coming up with the input sequences that validate properties is an arduous and a difficult task that needs the designer to know the complete details of the design. Let us take a simple example such as a FIFO component ignoring inputs when it is full. The input sequences for testing this property must ensure that the design transitions into the state whenthe FIFO is full. However, there are multiple paths that can take the design into the state where the FIFO is full. So, it is important to generate input sequences that cover the trivial and nontrivial cases. Moreover, it is crucial that the designer can direct the validation procedure for the design. In our work, we focus on validating the design by directing the generation of tests. It is not always simple to come up with these input sequences. 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 [2, 3]. Furthermore, there may be multiple paths (different input sequences) that transition the system to the same state. Authors in [2] provide input sequences by performing static analysis on the SystemC source and then use supporting tools for generating the sequences that serve as tests for the design. This approach is limited by the strength of the static analysis tools. In addition, it does not provide the flexibility for describing the reachable state of interest. 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, [2, 3] do 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 validation and in particular directed test-casegeneration 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 [4] or Spec# [5] and performing conformance 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) [6, 7] 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 [8–10] use SpecExplorer to put forward a development and verification methodology for SystemC. Except their focus is 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.

In this work, we present a model-driven methodology not only specifying and developing, but also validating system-level designs for SystemC [11]. Figure 1 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 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 an automaton from the specification. This automaton is then used to generate tests. 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. We use SpecExplorer’s exploration and test-case generation to create tests that are directed to reach interesting states of the design. While the input sequences are executed on the semantic model, the same inputs are exercised on the SystemC implementation model to validate whether the SystemC implementation model conforms to the AsmL specification.

1.1. *Main contributions*

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:

(i) we provide a formal semantics for the most recent version of SystemC’s DE simulation using ASMs as the semantic foundation,

(ii) we leverage Microsoft’s implementation of ASMs operational semantics to provide a simulation and debugging framework for the design specification described in AsmL,

(iii) we again leverage Microsoft’s existing algorithms for state space exploration in SpecExplorer to be effectively used for SystemC test generation for finding corner cases in the SystemC implementation,

(iv) we show how to generate wrappers for the SystemC library and SystemC implementation model to allow SpecExplorer to drive the SystemC simulation through C#,

(v) 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.

2. OUTLINE

The outline of this article is as follows: Section 3 discusses the necessary background and related work in fully understanding the proposed methodology and Section 4 presents the design flow. We present examples in Section 5 followed by our experience in using this approach in Section 6, and finally conclude in Section 7.

3. BACKGROUND AND RELATED WORK

3.1. *Abstract state machines*

Abstract State Machines (ASMs) [6, 7] 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

$$f(t_1, \dots, t_n) := t_0, \qquad (1)$$

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 $k = \{0, \dots, n\}$,

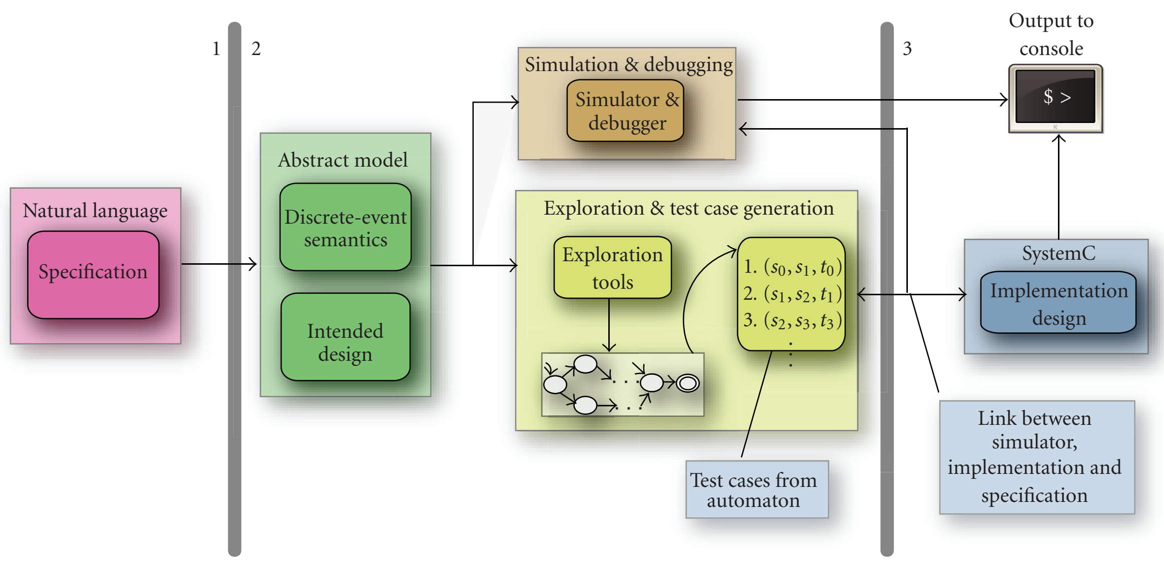


FIGURE 1: Methodology overview.

respectively. Upon the evaluation of v_k, the functions in the Updates set are evaluated. Hence, $f(v_1, \dots, v_n)$ is updated to v_0. A run of an ASM simultaneously fires 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 [6], synchronous ASMs, and distributed/asynchronous ASMs [6, 12]. Each of these variants posses certain descriptive capabilities. For example, Turbo-ASMs are ASMs appropriate for parallel and sequential composition, and recursive submachine calls.

3.2. SpecExplorer

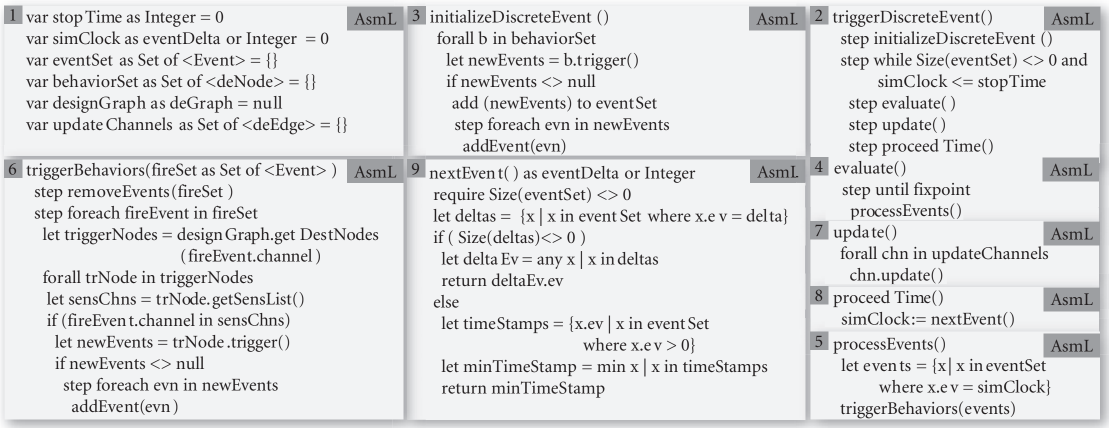
SpecExplorer is a specification exploration environment developed by Microsoft [4, 13] that is used for model-based specification and conformance testing of software systems. The language used for specification can be either AsmL [14, 15], a derivative of ASMs, or Spec# [5, 16].

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 [4].

3.3. 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 `discrete-event` class shown in Figure 2. 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 `trigger discrete-event` function marked with (2). Note that we use AsmL's parallel constructs such as the `forall` in `trigger behaviors` 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 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. We have made our entire semantics and examples downloadable via the web at [17].

The simulation begins with a function `start` that is not shown in Figure 2. 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 `trigger discrete-event`. This initializes the simulation by triggering every behavior in the system that in turn generates 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



```
1 var stopTime as Integer = 0                                    AsmL
  var simClock as eventDelta or Integer = 0
  var eventSet as Set of <Event> = {}
  var behaviorSet as Set of <deNode> = {}
  var designGraph as deGraph = null
  var updateChannels as Set of <deEdge> = {}
```

```
3 initializeDiscreteEvent ()                                     AsmL
   forall b in behaviorSet
    let newEvents = b.trigger()
    if newEvents <> null
     add (newEvents) to eventSet
      step foreach evn in newEvents
       addEvent(evn)
```

```
2 triggerDiscreteEvent()                                         AsmL
   step initializeDiscreteEvent ()
   step while Size(eventSet) <> 0 and
          simClock <= stopTime
     step evaluate( )
     step update( )
     step proceed Time()
```

```
6 triggerBehaviors(fireSet as Set of <Event> )                   AsmL
   step removeEvents(fireSet )
   step foreach fireEvent in fireSet
    let triggerNodes = designGraph.get DestNodes
                          (fireEvent.channel )
    forall trNode in triggerNodes
     let sensChns = trNode.getSensList()
     if (fireEvent.channel in sensChns)
      let newEvents = trNode .trigger()
      if newEvents <> null
       step foreach evn in newEvents
        addEvent(evn )
```

```
9 nextEvent( ) as eventDelta or Integer                          AsmL
   require Size(eventSet) <> 0
   let deltas = {x | x in event Set where x.e v = delta}
   if ( Size(deltas)<> 0 )
    let delta Ev = any x | x in deltas
    return deltaEv.ev
   else
    let timeStamps = {x.ev | x in event Set
                          where x.e v > 0}
    let minTimeStamp = min x | x in timeStamps
    return minTimeStamp
```

```
4 evaluate()                                                     AsmL
   step until fixpoint
    processEvents()
```

```
7 update()                                                       AsmL
   forall chn in updateChannels
    chn.update()
```

```
8 proceed Time()                                                 AsmL
   simClock:= nextEvent()
```

```
5 processEvents()                                                AsmL
   let events = {x| x in eventSet
          where x.e v = simClock}
   triggerBehaviors(events)
```

Figure 2: Discrete-event semantics using AsmL.

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.

3.4. 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 [4]. 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 [4, 18] 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.

A work on functional verification of SystemC models is proposed in [2]. In general, an FSM is generated by performing static analysis on the source code very much like [9] and this FSM is used to generate test sequences for the system under investigation. Authors of [2] 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 [2] 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. Full state space exploration is often times not practical and it is essential to be able to better direct the exploration. 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.

Recent work in design and verification methodologies for SystemC using SpecExplorer are presented in [8–10]. 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. Reference [9] also implements an AsmL specification of SystemC's simulation semantics [8], however, we were unable to obtain an executable version for us to reuse. 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.

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 [9], the same authors present algorithms for generating FSMs from SystemC source code. They 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 [8–10].

Authors of [3] 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 [2] that we provide a model-driven approach. The authors of [3] 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 [12] presented the ASM-based SystemC semantics that was later augmented for the newer versions of SystemC by [10]. However, the original ASM semantics in [12] did not present any support for test-case generation and validation from ASMs and was designed for an older version of SystemC.

4. DESIGN FLOW

The necessary components in employing this design flow are shown in Figure 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 3 with functions specific for a hardware FIFO component as our intended system (and from here on referred to it as that). We describe this example in detail in Section 5.

4.1. *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.

4.2. *SystemC modeling, simulation, and wrappers*

4.2.1. *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.

4.2.2. *Simulation*

The implementation model is simulated with the OSCI SystemC [1] simulator and if required, any other supporting tools may be used. This is standard practice when developing systems in SystemC.

4.2.3. *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.

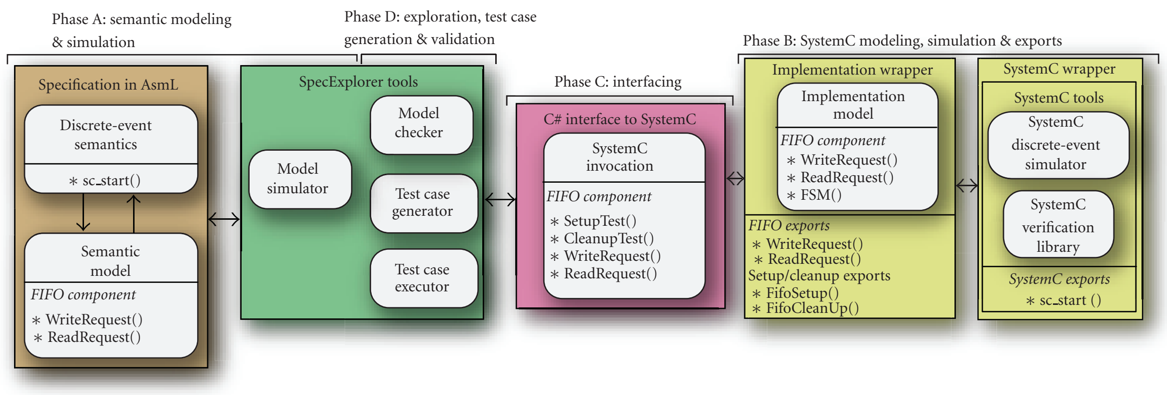


Figure 3: Modeling, exploration, and validation design flow.

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.

4.3. 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 performs 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, cleanup, requesting a write, and requesting a read. Each of these members invoke 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 argument 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.

4.4. 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.

5. 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.

5.1. FIFO

We elaborate on the FIFO component example in this section by presenting a block diagram in Figure 4. This is a simple FIFO component parameterized by `n` that represents the size

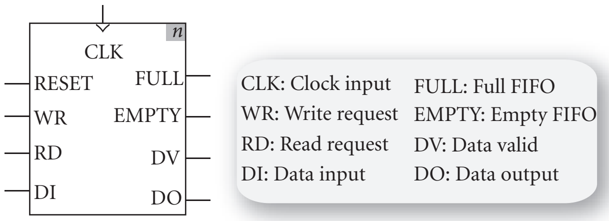


Figure 4: A FIFO component.

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 are 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.

5.1.1. 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 5. 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 are overlayed with a box in Figure 5. 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.

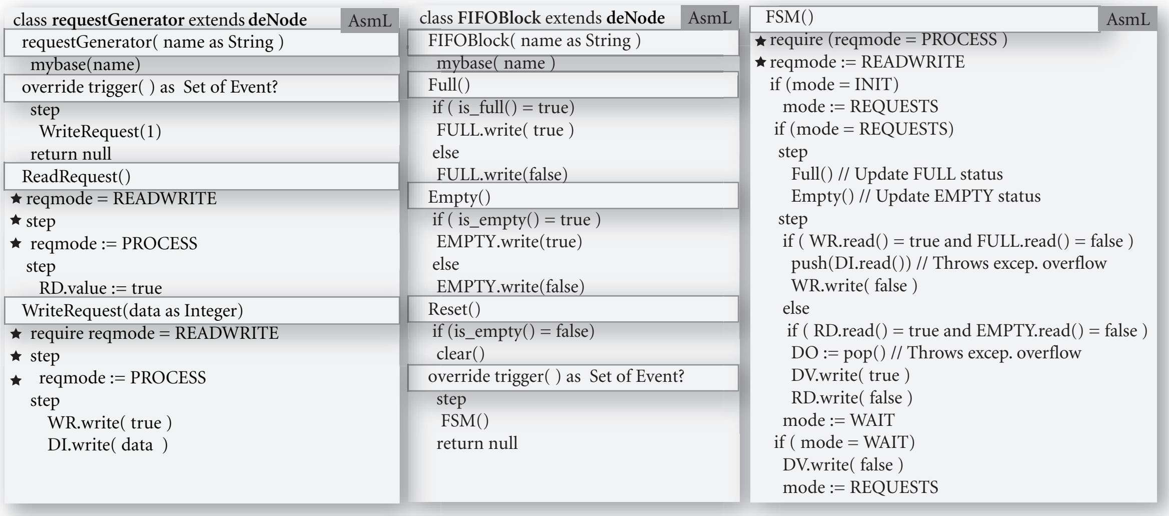
5.1.2. Exploration and test cases for specific properties

Notice in Figure 6 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 [19] 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. The automata for the FIFO are shown in Figure 7.

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 detailed discussion for the overflow property. We start by assuming that the $\star$ statements in Figure 5 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

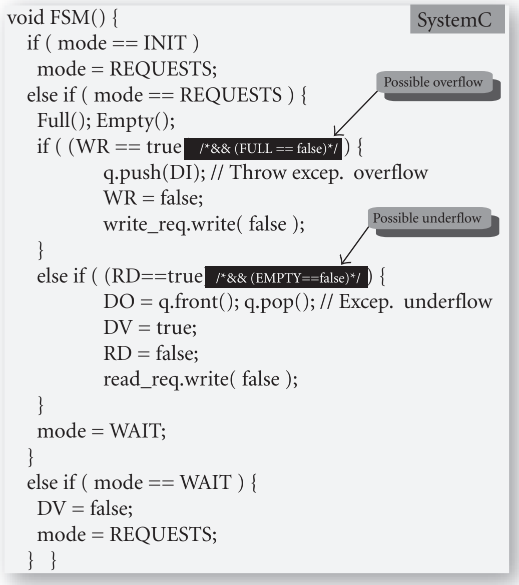


```
class requestGenerator extends deNode                AsmL
  requestGenerator( name as String )
    mybase(name)
  override trigger( ) as  Set of Event?
    step
      WriteRequest(1)
    return null
  ReadRequest()
★ reqmode = READWRITE
★ step
★  reqmode := PROCESS
    step
      RD.value := true
  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 5: FIFO specification in AsmL.


SystemC
Possible overflow
Possible underflow

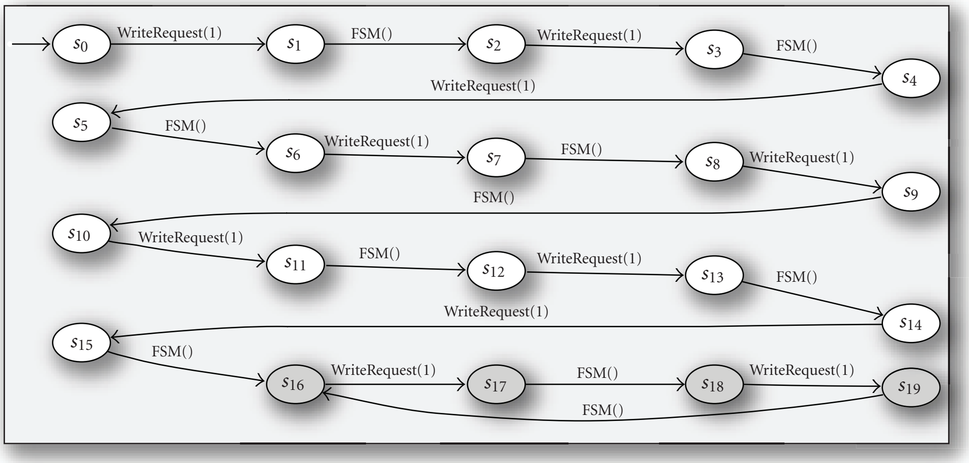
```
void FSM() {
  if ( mode == INIT )
    mode = REQUESTS;
  else if ( mode == REQUESTS ) {
    Full(); Empty();
    if ( (WR == true /*&& (FULL == false)*/ ) {
            q.push(DI); // Throw excep.  overflow
            WR = false;
            write_req.write( false );
    }
    else if ( (RD==true /*&& (EMPTY==false)*/ ) {
            DO = q.front(); q.pop(); // Excep.  underflow
            DV = true;
            RD = false;
            read_req.write( false );
    }
    mode = WAIT;
  }
  else if ( mode == WAIT ) {
    DV = false;
    mode = REQUESTS;
  }  }
```

Figure 6: SystemC FIFO FSM code snippet.

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.

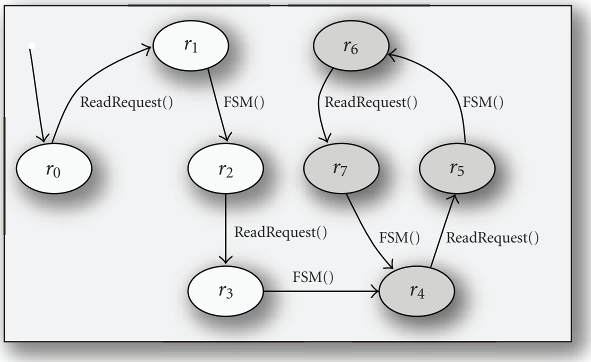
This above exploration configuration results in an invocation of each action from every state. This may be the correct automaton desired by the validation engineer, but suppose that for our investigation, we want to explicitly generate the automaton that first invokes `WriteRequest` followed by `FSM` 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 5. 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 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(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. These additions to the semantic model result in the automaton that is desired. Also remember that due to the internal state machine controlled by `mode`, it takes two invocations 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 is 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, backtracing 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



s_0 WriteRequest(1) s_1 FSM() s_2 WriteRequest(1) s_3 FSM() s_4
WriteRequest(1)
s_5 FSM() s_6 WriteRequest(1) s_7 FSM() s_8 WriteRequest(1) s_9
FSM()
s_{10} WriteRequest(1) s_{11} FSM() s_{12} WriteRequest(1) s_{13} FSM() s_{14}
WriteRequest(1)
s_{15} FSM() s_{16} WriteRequest(1) s_{17} FSM() s_{18} WriteRequest(1) s_{19}
FSM()

(a)



r_0 ReadRequest() r_1 FSM() r_2 ReadRequest() r_3 FSM() r_4 ReadRequest() r_5 FSM() r_6 ReadRequest() r_7 FSM()

(b)

Figure 7: Automata used for validation of FIFO.

error can also be duplicated. Evidently, the code fragment marked as a possible overflow situation in Figure 6 causes this erroneous behavior. Removing this yields in successful execution of the test cases.

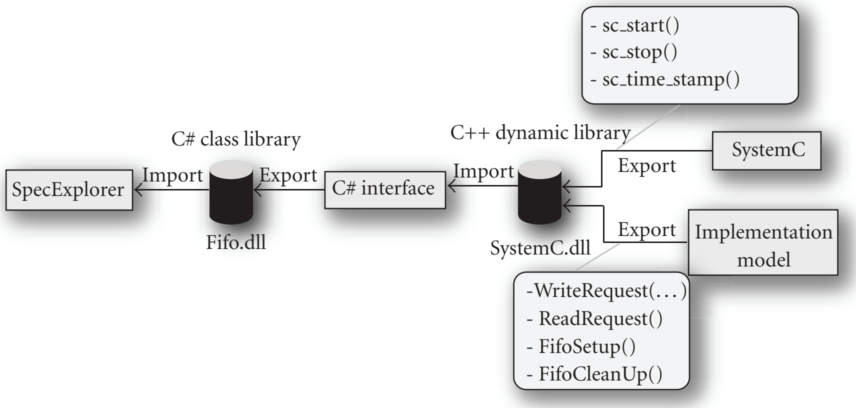
Generating test cases for validating the underflow property is done using a similar approach. The required steps in checking for underflow require the following: (1) adding `ReadRequest` and `FSM` as controllable actions in the exploration settings and (2) making the accepting state to be when the `EMPTY` state variable evaluates to true. The resulting automaton is shown in Figure 7(b).

Executing the test cases for this automaton again show 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 6 has the error marked as a possible underflow, the implementation throws an exception at this first successful read request. Once again, by uncommentingthe erroneous fragments, the test cases all succeed.

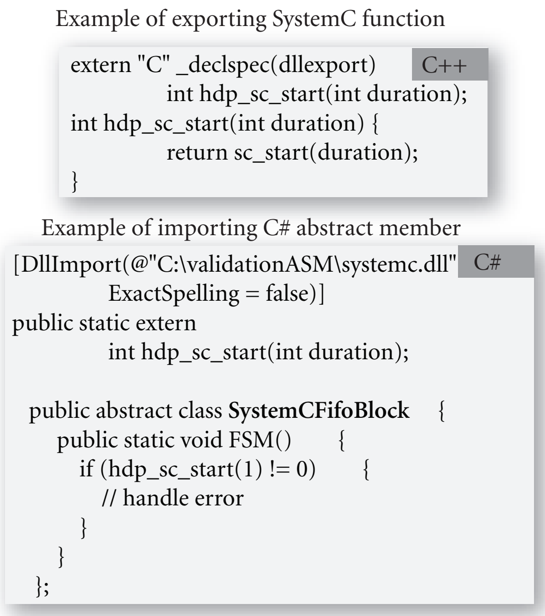
5.1.3. *Wrapper generation*

An integral part to the design flow is the construction of wrappers between SpecExplorer and SystemC. Figure 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 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 8(b), we show the definition of the `FSM` member in C#. Note that the class that



(a)


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)

Figure 8: Wrapper construction and export code snippets.

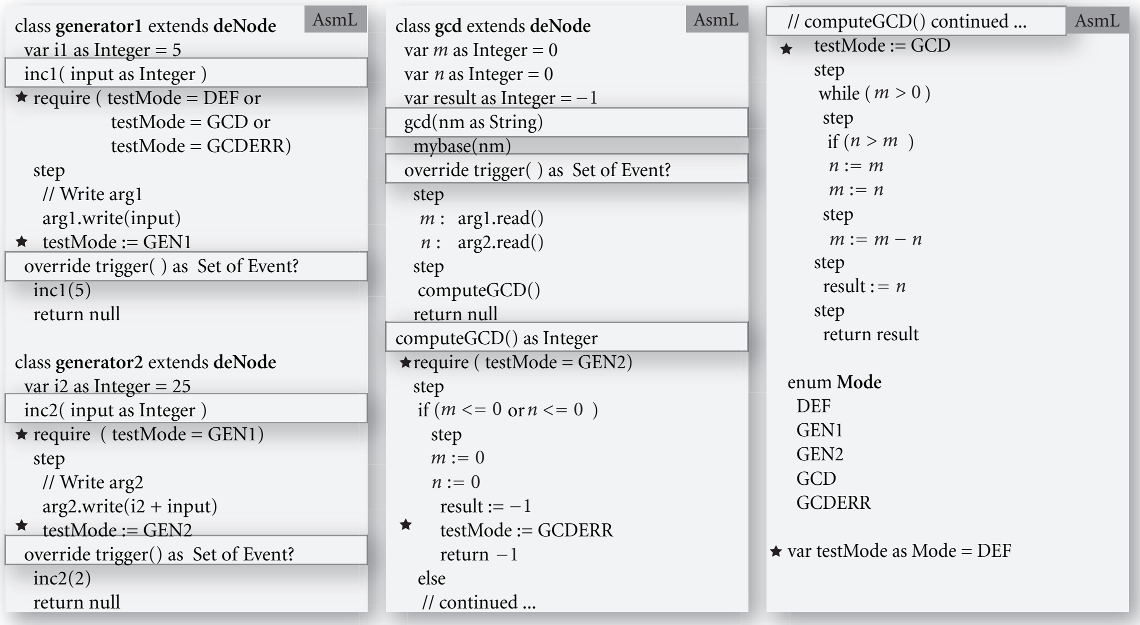
this member is contained in is an abstract class and that the types of the members must match those in the semantic model. This C# interface is then compiled as a class library that is called a reference in SpecExplorer. This reference is loaded into SpecExplorer so that test action bindings can be made.

5.2. 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 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 [17].

5.2.1. 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



```
AsmL
class generator1 extends deNode
  var i1 as Integer = 5
  inc1( input as Integer )
★ require ( testMode = DEF or
              testMode = GCD or
              testMode = GCDERR)
    step
      // Write arg1
      arg1.write(input)
★   testMode := GEN1
  override trigger( ) as  Set of Event?
    inc1(5)
    return null

class generator2 extends deNode
  var i2 as Integer = 25
  inc2( input as Integer )
★ require  ( testMode = GEN1)
    step
      // Write arg2
      arg2.write(i2 + input)
★     testMode := GEN2
  override trigger() as  Set of Event?
    inc2(2)
    return null
```

```
AsmL
class gcd extends deNode
  var m as Integer = 0
  var n as Integer = 0
  var result as Integer = −1
  gcd(nm as String)
    mybase(nm)
  override trigger( ) as  Set of Event?
    step
      m :   arg1.read()
      n :   arg2.read()
    step
      computeGCD()
    return null
computeGCD() as Integer
★require ( testMode = GEN2)
  step
   if (m <= 0 or n <= 0 )
     step
     m := 0
     n := 0
      result := −1
★     testMode := GCDERR
      return −1
   else
    // continued ...
```

```
// computeGCD() continued ...            AsmL
★   testMode := GCD
    step
     while ( m > 0 )
      step
       if (n > m  )
        n := m
        m := n
      step
        m := m − n
    step
      result := n
    step
      return result

  enum Mode
    DEF
    GEN1
    GEN2
    GCD
    GCDERR

★ var testMode as Mode = DEF
```

Figure 9: GCD AsmL specification.

interested in validating. They are as follows:

(i) the intended operation of the GCD computation component when encountering invalid input data;

(ii) simply validating when the input sequence consists 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 nonnegative 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 11.

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 10.

5.3. 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 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 a 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.

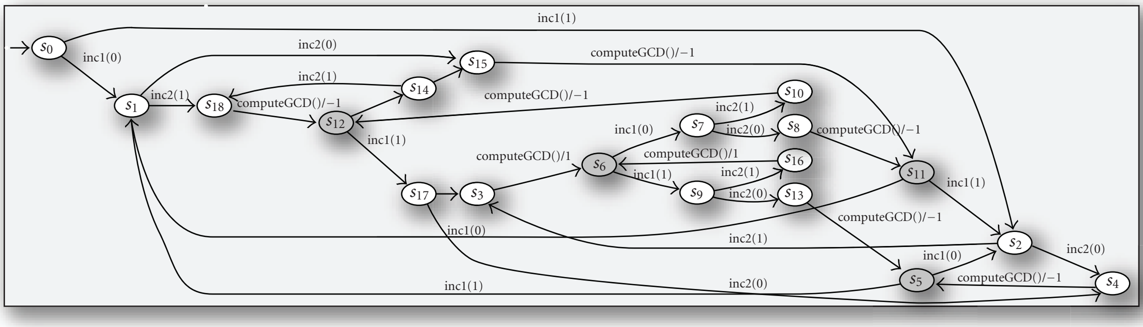


Figure 10: Automaton for validating GCD with 0 or 1 as inputs.

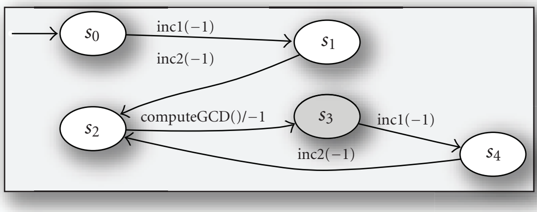


Figure 11: Automaton for validating invalid inputs.

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.

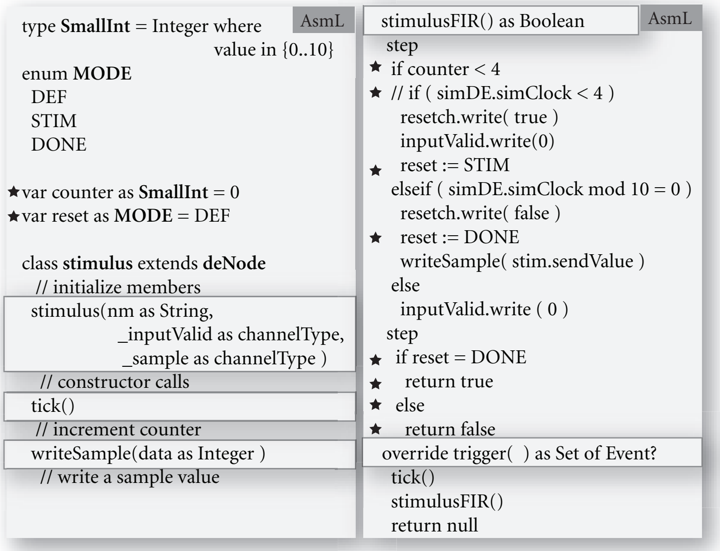
The automaton generated from the above exploration configuration is shown in Figure 12(b). Executing the test cases generated for this automaton show that the reset is indeed held high for four cycles thus showing conformance between the semantic and implementation models. The full source for the semantic model, implementation model, and the wrappers are available at [17] along with the other examples.

6. OUR EXPERIENCE

The first phase of semantic modeling is intuitive and simple; even for traditional hardware designers. This is because the additional DE simulation semantics in ASM makes it simple for designers already familiar with SystemC or any other DE simulation environment to describe their semantic models. More importantly, hardware designers do not need to know the complete formal semantics of ASMs, but rather only the same concepts that already exist in traditional hardware description languages such as concurrent statements. The modeling paradigm and extensibility of the DE semantics allows for user-defined channels and modules. The semantics also take into account the nondeterminism mentioned in the SystemC IEEE 1666 standard. This is done by using parallel constructs in AsmL such as the `for all`. 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 [12] 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 [20], and thus refrain from extending our semantics for the sake of clarity. The simulation and debugging of the semantic models are 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 most challenging aspect of this methodology was the exploration of the semantic model. This is because, similar to other model-checking engines, the automata generation can suffer from state space explosion. A user can specify state count and transition count constraints in order to time-out the automata generation engine indicating that further tweaking of the semantic model is required. Tweaks to the semantic model are called exploration techniques [4] such as state grouping, state filtering, and parameter selections. These techniques help in abstracting the semantic model to focus on the states and transitions that interest the designer, resulting in a reduction of the state space. Therefore, large designs with a large state space can be incrementally abstracted using these exploration techniques, without having to change much of the semantic model. We acknowledge that the effort in being able to generate the automaton for large designs is more involved, which requires the application of rigorous state space pruning techniques. This however, requires a thorough understanding of how automata are generated from AsmL specifications. Our experience suggests that after exploring on a few examples, the techniques and capabilities of SpecExplorer can be understood clearly; thus making the exploration



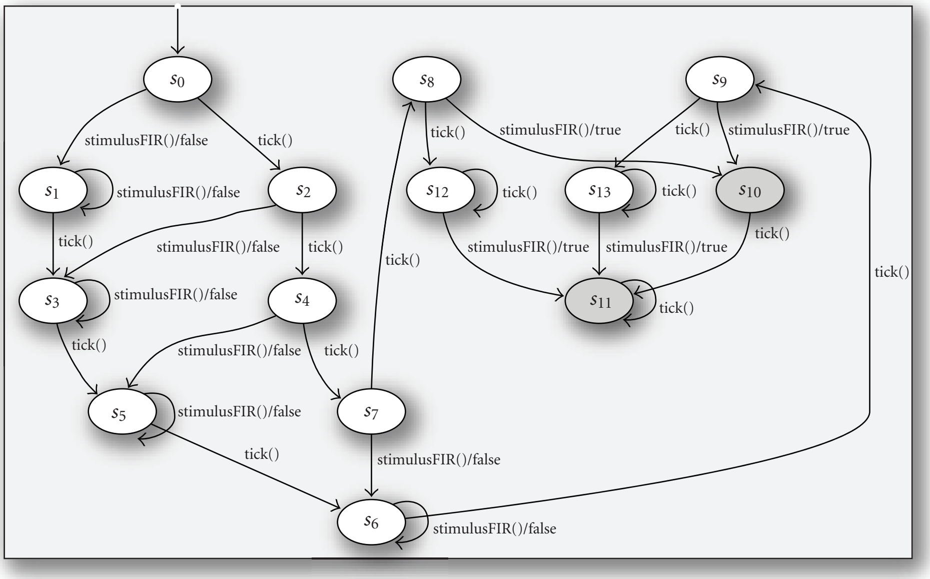
```
AsmL
type SmallInt = Integer where
                      value in {0..10}
enum MODE
  DEF
  STIM
  DONE

★var counter as SmallInt = 0
★var reset as MODE = DEF

class stimulus extends deNode
  // initialize members
  stimulus(nm as String,
           _inputValid as channelType,
           _sample as channelType )
  // constructor calls
  tick()
  // increment counter
  writeSample(data as Integer )
  // write a sample value
```

```
stimulusFIR() as Boolean                    AsmL
  step
★  if counter < 4
★  // if ( simDE.simClock < 4 )
     resetch.write( true )
     inputValid.write(0)
★    reset := STIM
   elseif ( simDE.simClock mod 10 = 0 )
     resetch.write( false )
★    reset := DONE
     writeSample( stim.sendValue )
   else
     inputValid.write ( 0 )
  step
★  if reset = DONE
★    return true
★  else
★    return false
override trigger( ) as Set of Event?
   tick()
   stimulusFIR()
   return null
```

(a)



(b)

Figure 12: FIR example.

much simpler for a designer. After successfully obtaining an automaton, the test sequence generation is quick and simple. In fact, generating the simulation sequences corresponds to path traversals on the automaton. This step is automated and takes little time when using the existing algorithms for traversal in SpecExplorer.

Currently, the wrappers are generated manually. The information that these wrappers require are the methods that change the states of interest. Since these methods are already specified in SpecExplorer, we can use this information to automatically generate the C# wrapper with the action binding members. We can do the same for exporting the method functions of the implementation models. However, the SystemC library wrapper only needs to be done once and we provide that. The effort in creating the C# and the SystemC wrappers can be significantly reduced when this step is automated.

7. CONCLUSIONS

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 detail 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. There is an associated learning curve in knowing how to use SpecExplorer and its methods for state space pruning and exploration. 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.

ACKNOWLEDGMENTS

We acknowledge the support received from the NSF CAREER Grant CCR-023794 and the NSF NGS program Grant ACI-0204028.

REFERENCES

[1] OSCI, SystemC and SystemC Verification, http://www.systemc.org/.

[2] 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.

[3] D. Kroening and N. Sharygina, "Formal verification of SystemC by automatic hardware/software partitioning," in *Proceedings of the 3rd ACM and IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE '05)*, pp. 101–110, Verona, Italy, July 2005.

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

[5] Microsoft Research, Spec#, http://research.microsoft.com/specsharp.

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

[7] Y. Gurevich, "Evolving algebras 1993: Lipari guide," in *Specification and Validation Methods*, pp. 9–36, Oxford University Press, New York, NY, USA, 1995.

[8] A. Gawanmeh, A. Habibi, and S. Tahar, "Enabling SystemC verification using abstract state machines," Tech. Rep., Department of Electrical and Computer Engineering, Concordia University, Montreal, Quebec, Canada, 2004.

[9] A. Habibi, H. Moinudeen, and S. Tahar, "Generating finite state machines from SystemC," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '06)*, vol. 2, pp. 76–81, Munich, Germany, March 2006.

[10] A. Habibi and S. Tahar, "Design and verification of SystemC transaction-level models," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 14, no. 1, pp. 57–68, 2006.

[11] H. D. Patel and S. K. Shukla, "Model-driven validation of SystemC designs," in *Proceedings of the 44th ACM/IEEE Design Automation Conference (DAC '07)*, pp. 29–34, San Diego, Calif, USA, June 2007.

[12] W. Mueller, J. Ruf, D. Hoffmann, J. Gerlach, T. Kropf, and W. Rosenstiehl, "The simulation semantics of SystemC," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '01)*, pp. 64–70, Munich, Germany, March 2001.

[13] M. Barnett, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes, "Towards a tool environment for model-based testing with AsmL," in *Proceedings of the 3rd International Workshop on Formal Approaches to Testing of Software (FATES '03)*, A. Petrenko and A. Ulrich, Eds., vol. 2931 of *Lecture Notes in Computer Science*, pp. 264–280, Montreal, Quebec, Canada, October 2003.

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

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

[16] M. Barnett, K. R. M. Leino, and W. Schulte, "The Spec# programming system: an overview," in *Proceedings of the International Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS '04)*, pp. 49–69, Marseille, France, March 2004.

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

[18] C. Campbell, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillman, and M. Veanes, "Model-based testing of object-oriented reactive systems with spec explorer," Tech. Rep. MSR-TR-2005-59, Microsoft Research, Cambridge, UK, 2005.

[19] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes, "Generating finite state machines from abstract state machines," in *Proceedings of the ACM International Symposium on Software Testing and Analysis (ISSTA '02)*, pp. 112–122, ACM Press, Roma, Italy, July 2002.

[20] S. A. Sharad and S. K. Shukla, *Correctness Preserving Transformations of System Level Models for Efficient Simulation, Formal Methods and Models for System Design: A System Level Perspective*, Kluwer Academic Publishers, Norwell, Mass, USA, 2005.