Design for Verification of SystemC Transaction Level Models

Ali Habibi and Sofiène Tahar
Department of Electrical and Computer Engineering
Concordia University
1455 de Maisonneuve, West
Montreal, Québec, H3G 1M8, Canada
Email: {habibi,tahar}@ece.concordia.ca

Abstract

Transaction level modeling allows exploring several SoC design architectures leading to better performance and easier verification of the final product. In this paper, we present an approach to design and verify SystemC models at the transaction level. We integrate the verification as part of the design-flow. In the proposed approach, we first model both the design and the properties (written in PSL) in UML. Then, we translate them into an intermediate format modeled with Abstract State Machines (ASM). The ASM model is used to generate an FSM of the design including the properties. Checking the correctness of the properties is performed on-the-fly while generating the state machine. Finally, we translate the verified design to SystemC and map the properties to a set of assertions (as monitors in C#) that can be re-used to validate the design at lower levels through simulation. We illustrate our approach on two case studies including the PCI bus standard and a generic Master/Slave architecture from the SystemC library.

1. Introduction

SystemC [10] is a relatively new system level language proposed to overcome the problem of the growth in complexity and size of systems combining different types of components. SystemC meets the need for a system level language that can fill the gap between hardware description languages (HDLs) and traditional software programming languages. For that reason, it is more important to focus on defining methodologies and techniques for SystemC transactional models a finite set of architecture components (memories, CPUs, etc) communicate among each other over shared resources (buses).

Until recently, modeling architectures required pin-level hardware descriptions, typically RTL level. Great effort is required to design and verify the models, and simulation at this level of detail is tediously slow. Transaction level modeling is eventually the best solution. In addition to the design's approach issue, the verification of a SystemC model is a serious bottleneck in the design cycle. Going further in complexity and considering hardware/software systems will be out of the range of nowadays used simulation based techniques.

In conventional HDL designs, defining the system's properties and specification is itself a problem. In order to provide an efficient language to write assertions and properties, the Accellera organization proposed the Property Specification Language (PSL) [1], which addresses the lack of information about properties and design characteristics. Nevertheless, defining the language itself is not enough to improve both the design and verification flows.

In this paper, we first provide a methodology to design SystemC transactional models starting from the UML level where a more precise specification of the system and its properties are defined. We used a modified sequence diagram representation to capture more precise properties description. The UML model is then mapped to an Abstract State Machines (ASM), a formal specification method for software and hardware systems that has become successful for specifying and verifying complex systems. ASMs provide features to capture the behavioral semantics of programming and modeling languages where large systems are modeled on a high level of abstraction allowing easier validation and verification operations. We considered AsmL language [8] which was developed at Microsoft because: (1) it effectively supports specification and rapid prototyping of different kinds of models; (2) it includes a tester that can be used to generate finite state machines (FSMs) and test cases.

To enable the integration of both the model and the properties at the ASM level, we embedded the PSL semantics in ASM. At this level, it is possible to verify these properties using model checking. For instance, we encode the property's evaluation in every state which enables evaluating its correctness on-the-fly while executing the FSM generation algorithm (part of the AsmL tool). An incorrect property detection stops the reachability algorithms and outputs a sub-portion from the complete FSM which represents a complete scenario for a counter-example. Eventually, not all the properties can be verified due to the state explosion problem. For this reason, we complement our verification methodology by integrating the properties as assertion monitors in the final SystemC design. We compile the PSL property (in ASM) to a C# while the SystemC model (in ASM) is translated SystemC. Both codes are then combined to form a single model enabling the verification of the assertions by simulation.

The rest of this paper is organized as follows: Section II presents the proposed design methodology. Section III discusses the proposed verification approach. Section IV illustrates our approach on two bus structure case studies. Finally, Section V discusses the related work and concludes the paper.

1530-1591/05 $20.00 © 2005 IEEE

2. Design Methodology

Our design methodology, as displayed in Figure 1, includes two parallel paths concerning the design and its properties. We model the design in the classical way a C++ design is modeled using UML (i.e., using use cases, class diagrams, etc.) Then, we translate the UML model to ASM in order to perform model checking of certain properties. These latter are extracted from the UML sequence diagram and encoded in the PSL syntax. The verification process ends to: (1) a completion either with a success or failure of the property; or (2) a state explosion. The UML update and UML to ASM translation tasks are repeated until all the properties passes with success either succeeds or do not complete. Then, we compile the PSL properties into a set of C# classes, using the AsmL tool to be used as assertion monitors. The design in ASM is, from the other side, translated to C++ (SystemC model) and co-integrated with the assertions for verification by simulation.

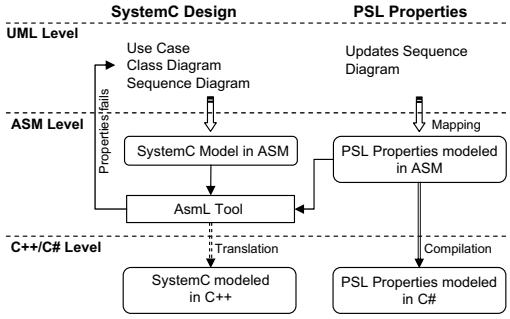

Figure 1. Design and Verification Methodology.

2.1. Modeling PSL Properties

PSL is an implementation independent language to define properties. PSL is a hierarchical language, where every layer is built on top of the layer below. This approach allows the expressing of complex properties from simple primitives.

2.1.1. UML Model Using UML as a high level of abstraction for design showed a lot of success when applied to Software. Main proposals consider either to use UML as new system level design [2] or as top layer in combination with existent languages (such as SystemC)[11]. Nevertheless, these proposals neglected totally considering the properties of the system (PSL like properties in particular) while sequence diagrams for example includes very useful information to set transaction properties for TLM in particular.

Unfortunately, sequence diagrams do not allow a direct mapping to PSL due to two reasons: (1) the complexity of the PSL property which may include temporal operators; and (2) the need for instantiation in the PSL. In fact, PSL was defined for a real instances from the design formed from objects while the sequence diagram considers only classes. For these facts, UML will not present completely and precisely all PSL property. However, it can be used to provide a general skeleton of the property that could be refined and instantiated at the ASM level.

In order to make the UML sequence diagram more adequate for PSL representation, we introduced the following operators:
Clocks: we use the operator to specify the clock that activates the current action (if there exist one).
Number of cycles: every action can be include the information about after how many cycles the method is start executing (for e.g., *Mtd[5]()* says that the method *Mtd* is executed for exactly 5 consecutive cycles).
Temporal operators: these includes operators specifying if the method will be always executed (A), eventually executed (E), executed Until a condition is fulfilled (U), etc. These, in fact, represent a mapping to the PSL temporal operators (second layer of PSL).
Sequence operations: includes information about the order of executing certain sequences (for e.g., *next*, *prev* etc.)
Text output: refers to a message that is displayed in case the method fails. This is included in PSL to track the progress of the assertion based verification.
Method duration: certain methods are supposed to execute for a certain number of cycles (for e.g., reading for memory may take 4 cycles). So, we added an operator $ to specify such an information.

Figure 2 gives an example of a sequence diagram describing a PSL property saying that if a bus sends a new request, then in the next cycle the arbiter will be notified and will make the arbitration. In the third cycle, the Master starts sending. The bus is released in the 4 cycle and a notification will be sent, eventually, by the slave to the bus who will forward it in the next cycle to the Master.

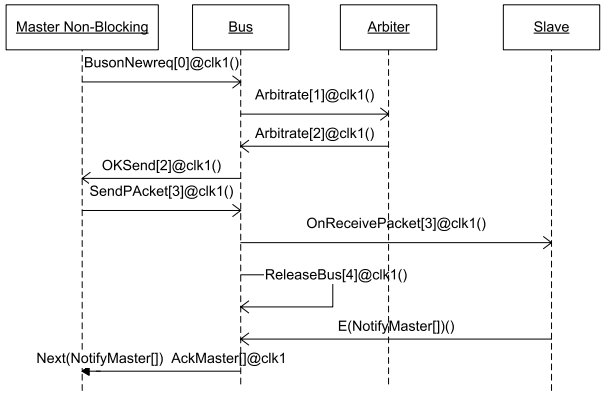

Figure 2. Example of a Modified UML Sequence diagram.

When mapping to ASM the UML sequence diagram needs to be instantiated according to the design objects. For instance we need to specify, for example, that the notification must be to the original master and not to all the masters.

2.1.2. ASM Model Abstract State Machines (ASM) [7] is a formal specification method for software and hardware systems that has become successful for specifying and verifying complex

systems. There are many languages that have been developed for ASMs, the recent one is AsmL [8] which was developed at Microsoft. The AsmL tester can also be used to generate finite state machines (FSMs) and test cases. We notice that AsmL is language that supports object-oriented modeling at higher level of abstraction in comparison to C++ and Java. So, it is not a graphical representation but a programming language that can be used to describe system models either as abstract state machines or programming models.

There are two ways to embed PSL properties into the design, either into the design code itself or by adding them as external monitors. We adopted the first approach, where all the parameters of PSL properties are defined as objects. The objective of the embedding is to reuse PSL properties, as embedded in ASM, at lower design levels since the AsmL tool can automatically compile them into a C# or .NET code, which can be compiled and executed with the concrete SystemC level or as a stand-alone module.

PSL properties are defined in a hierarchical way inspired from the hardware design modular concept. For this reason we defined the embedding in a similar structure, where all the components are defined as objects and every PSL layer *extends* its lower layer using the inheritance feature of AsmL.

Boolean Layer:

This layer is the basic layer of PSL. Even though it is called *Boolean layer*, it includes types other than Boolean such as integers and bit vectors. We embedded this layer in ASM by defining classes for all types and expressions including their methods. Our embedding is based on the semi–formal semantics presented in the reference manual [1], and the formal semantics definition in HOL [4].

The embedding of the PSL Boolean layer mainly includes:
(1) Expression type class includes the basic 5 types: *Boolean*, *PSLBit*, *PSLBitVector*, *Numeric* and *String*. Both *Boolean* and *String* types are directly inherited from the ASM's $AsmL.Boolean$ and $AsmL.String$, respectively.
(2) PSL Expressions includes constructing properties using the implication and equivalence operators.
(3) PSL Built Functions include all the functions defined by PSL to operate at the Boolean layer. We distinguish here two methods: a method that provides the previous values of a variable (e.g., *prev()*) and a method that provides the future values of a variable (e.g., *next()*).

Temporal Layer:

The most important part of this layer is the SERE feature, which embedding mainly includes:
(1) Sequential Expressions, where a SERE is defined as an AsmL sequence of Boolean. It offers several operations to construct, manipulate and evaluate the SERE expression. *(2) Properties* including the operations necessary to create properties from sequential expressions. It also controls when and how the sequence is to be verified (i.e., the property "verify the sequence is true after n states" is defined as *PSL_Property.EvaluateNext(n)*).

Figure 3 shows the example of the *PSL_SERE.Evaluate()*, which checks if a sequence is true in a certain path. This method is activated according to an *INIT* signal that must be set by the property.

Verification Layer:

```
class PSL_SERE
  // Memebers
  var m_size as Integer = 0   var m_seq as Seq of Boolean
  var m_cycle as Seq of Integer  var m_actualState as Integer = 0
  var m_evaluation as SERE_Evaluation = NOT_STARTED
  // Methods
    public Evaluate() as SERE_Evaluation
     require m_evaluationState = INIT
     if(me.m_seq(m_actualState) = false)
       m_evaluation := FAILED  return FAILED
     else
       if m_actualState = m_size
         m_actualState := m_actualState + 1 return IN_PROGRESS
       else
         m_actualState := 0  return SUCCEEDED
```

Figure 3. Embedding PSL SERE in ASM.

This layer is intended to tell the verification tool how to perform the verification process. It allows to construct assertions from properties and to specify relations between them. The embedding mainly includes:
(1) Verification Directives to specify how the property will be interpreted (assertion, requirement, restriction or assumption). This class extends the *PSL_Property* class from the temporal layer.
(2) Verification Unit is a compact way to include several properties together. The embedded class includes several operations to add/remove and update the unit's list of properties.

Modeling Layer:

This layer is not used in our verification approach since it is intended for VHDL and Verilog flavors of PSL. So we did not consider it in our embedding.

2.2. Modeling SystemC

SystemC is built on standard C++. The core language consists of an event-driven simulator as the base. It works with events and processes. The other core language elements consist of modules and ports for representing structures. Interfaces and channels are used to describe communications. SystemC provides data-types for hardware modelling and certain types of software programming as well.

2.2.1. ASM Model The design model at the ASM level is purely object-oriented where every class includes a set of parameters and methods. The particularity of this model resides in the fact that it will be used to generate an FSM using the reachability algorithm embedded AsmL tool. So, a specific style of programming is required in addition to a precise configuration of the algorithm. Before going further into the details of the ASM model, we will discuss the FSM generation algorithm which will give a better idea about the requirements of the ASM model.

The FSM generation algorithm requires as input: domains, methods, actions and variables (optional inputs are filters, action groups and properties). The transitions in the FSM are the method calls (including argument values) in the test sequences. The methods in the model program that appear in the transitions are called actions. The states in the FSM are determined by the values of selected variables in the model program, called state variables. The algorithm generates the FSM by executing

the model program in a special execution environment, keeping track of the actions it performs and recording the states it visits. This process is called *exploration*. Usually a model program implies so many states and transitions that it is not feasible to include them all in the FSM, so it is a must to limit the number of states and transitions that the tool explores.

The final FSM will, according to the algorithm's configuarion options, represent only a portion – an under-approximation – of the huge FSM that would result if the model program could be explored completely. Critical information to ensure the correctness of certain properties concerning identifying the actions and state variables in the FSM. Domains, finite collections of values from which method arguments are taken, are defined in order enable better coverage on particular issues. Filters express stopping conditions that limit exploration (used to stop the FSM generation if a property fails, for e.g.).

For a model M including a set of classes, $C = \{c_1, \ldots, c_n\}$, where n is the total number of classes in M. For every class c_i in C, we denote its set of methods by c_i^{mth} and the set of members by c_i^{mem}. We defined a set of rules (called R_{FSM}) to guarantee the generation of an FSM representing a portion of the complete system's FSM; these include:

Rule R^1_{FSM}**:** For every class c_i in C, we have to define a list of instantiations of the class. This ensures that the algorithm will not through an exception.

Rule R^2_{FSM}**:** The firstly executed method in the design must verify that all the objects from the class domains were correctly instantiated. This ensures that the algorithm will not misbehave.

Rule R^3_{FSM}**:** For every class c_i in C, every method in c_i^{mem} must include a list of *pre-conditions* to specify when the algorithm considers this method in the exploration process. This ensues that in every state we only explore the involved methods.

Rule R^4_{FSM}**:** For every class c_i in C, domains for all members in c_i^{mem} must be inherited from AsmL types and restricted to the possible values the system can accept (in particular for inputs). This will allow exploring known types and limits the risks of state explosion.

The optimal scenario is to explore all the methods and domains in the model; nevertheless, this is not possible all the time due to the state space explosion. For this reason, working carefully the domains and the set of actions is the very critical path in the FSM generation process. For illustration purpose, Figure 4 shows a generic ASM model with a method including a precondition (denoted by the *require* keyword) setting that the method needs the system to be initialized ($SystemInit = true$) and that it has both variables *m_gnt* and *m_req* set to *false* before it can be executed. Such a conditions define strictly at which state the system can execute a particular set actions.

2.2.2. Translation to C++ Once the ASM model verified using the properties describing its behaviour, we translate it to SystemC according to a set of rules to ensure that the final SystemC model preserves the original ASM code properties. The transformation is purely syntactical, it is performed to certain rules (that we call R_{C++}) that could be summarized in the following:

Rule R^1_{C++}**:** "Basic types": ASM basic types are all mapped to their equivalent SystemC types (e.g. *Integer* to *int*, *Byte* to *unsigned char*, etc.). AsmL includes the same types as C++ which are used for SystemC also.

```
class PCI_Arbiter
  private var m_ActiveMaster as Integer = -1
  private var m_req as Boolean = false
  private var m_gnt as Boolean = false
  public PCI_Arbiter()
  public PCI_ArbiterUpdate_m_req()
    require (SystemInit = true) and me.m_gnt = false and me.m_req = false
    me.m_ActiveMaster := min id | id in Masters_Range where
                                  (MASTERS(id).m_req =  true)
    me.m_req := true
```

Figure 4. An Example of an ASM Model.

Rule R^2_{C++}**:** "Class Translation": this includes two separate rules for variables and methods:

Rule $R^{2.1}_{C++}$*:* "Class Members": are translated into SystemC signals having the same basic type. For e.g., *var m_val as Integer* is translated to *sc_signal<int> m_val*.

Rule $R^{2.2}_{C++}$*:* "Class Methods": in ASM contain two parts first one defining the post-/pre-conditions for its execution and the method itself. The first part is integrated in the SystemC module's *constructor*. For instance, a method *Send* defined in ASM with the following pre-condition *require clk=true* is inserted in the SystemC module constructor area as "*SC_THREAD(Send); sensitive << clk;*". The method itself is integrated as it is in the SystemC module (we just modify the basic types according to the Rule 1).

Rule R^3_{C++}**:** "Global Modules": are integrated in the SystemC's main procedure *sc_main*. The naming mapping is used to link different modules together.

3. Verification Methodology

The verification process is decomposed into two parts: (1) by model checking at the ASM level; and (2) by assertion based verification at the C++/C# level.

3.1. Model Checking

PSL properties are embedded in ASM as assertions, the assertion here means the validity of the property. It provides a unique view of the property in every system's state. It also simulates the design with the property as a monitor. We build the assertion starting from basic Boolean components, sequences, and then verification units. We encapsulate sequences in the verification unit as an assertion which is embedded in the design. Given a set of Boolean items $x_1, x_2, \ldots, x_n$, and $y_1, y_2, \ldots, y_m$ belonging to the Boolean layer, and the sequences, S_1 and S_2 belonging to the temporal layer, we can define: $S_1 = \{x_1, x_2, \ldots, x_n\}$, and $S_2 = \{y_1, y_2, \ldots, y_m\}$ and then use assertions to check any PSL operation between S_1 and S_2 such as $S_1\ OP\ S_2$, where OP is a PSL operator (e.g., implication ($\Rightarrow$), or equivalence ($\Leftrightarrow$)). The assertion is built as follows:

1. Add all the Boolean items to the sequences:
   $\forall\, i\ in\ 1\ to\ n:\ S_1.AddElement(x_i)$
   $\forall\, j\ in\ 1\ to\ m:\ S_2.AddElement(y_i)$

2. Create the property: $P := S_1\ OP\ S_2$

3. Define the *verification unit* as an assertion, say A, that includes the above property: $A.Add(P)$

This property is embedded in every state in the FSM generated by the AsmL tool and is represented by two Boolean state variables *P_eval* and *P_value* (saying, respectively, if the property can be evaluated and the value of the property in the current state). A violated property is detected once *P_eval = true and P_value = false*. We set the previous condition as filter for the FSM generation algorithm. This way the generation stops when an error is detected. The generated portion of the state machine, at this point, can be used to identify the problem through a scenario of a counter-example. For multiple properties, the filter is set as conjunction of all the conditions for the separate properties. This technique minimizes radically the number of the state variables (the FSM size and its generation time). A correct verification process results on the generation of the system's FSM (according the configuration file constraints).

3.2. Assertion Based Verification

We target to add the assertion as an external monitor to the SystemC design. We consider three steps:
(1) Updating the SystemC design to interface to the assertion.
(2) Generating the assertion (in C#) from its ASM description.
(3) Integrating the assertion in the design.

The translation to C# of the PSL assertions embedded in ASM is a matter of compilation using the AsmL tool. Most of the effort is spent in updating the SystemC design to get it connected the assertion monitor. For instance, we validate the assertion syntactically by generating the list of its involved variables. Then, we perform a type check to make sure the variables are well instantiated in the SystemC design. For instance, the signals (variables) that are used in the assertion must be seen as external signals so that they can be input to the assertion monitor. So, modify the SystemC design to make the required variables visible to the monitor. This transformation does not affect the behavior of the code as it will only be accessed in a read–only mode.

Once the design is updated, we add the required instantiation of the assertion to bind it to the existing SystemC design modules. The assertion monitor, acting as part of the design, can do the following: (1) stop the simulation when the assertion is fired; (2) write a report about the assertion status and all its variables; and (3) send a warning signal to other modules (if required). We note that the internal code of the assertion is C# so the designer can update it or do any other functionalities that can be coded in C#.

4. Experimental Results

In order to illustrate the proposed design and verification methodology, we considered two models: (1) PCI (Peripheral Component Interconnect) [6] Local Bus standard; and (2) an extension of the Master/Slave Bus structure provided by the SystemC distribution [10]. Both models include ceratin properties, such as liveness, that cannot be verified using simulation which requires using formal verification techniques such as model checking. Moreover, we aim evaluating the performance of the overall approach according to the system's size, which can be performed by varying the number of masters and slaves. The experiments were conducted on a Pentium IV processor (2.4 GHz) with 512 MB of memory.

Number of		**Model Checking**			**Simula-tion**
		CPU	Number of FSM		
Mas.	Sla.	Time (s)	Nodes	Trans.	δ (ns)
1	1	2.31	20	25	24.31
1	2	2.93	39	53	29.32
3	1	26.01	236	341	29.76
2	2	26.84	293	449	30.89
2	3	101.37	658	1117	32.74
3	2	574.18	1881	3153	34.03
3	3	6836.01	6346	12097	36.82

Table 1. PCI Bus: Model Checking and Simulation Results.

4.1. Models Description

PCI boasts a 32-bit data path, 33MHz clock speed and a maximum data transfer rate of 132MB/sec. Each PCI master has a pair of arbitration lines that connect it directly to the PCI bus arbiter. In the PCI environment, bus arbitration can take place while another master is still in control of the bus. Data is transferred between an initiator which is the bus master, and a target, which is the bus slave. PCI supports several masters and slaves and allows stopping transactions.

The SystemC Master/Slave bus represents a more generic bus structure including a set of Masters, a set of slaves an arbiter and a shared bus. The arbiter is responsible for choosing the appropriate master when there is more than one connected to the bus. There are two possible modes for the bus: (1) *Blocking Mode*, where data is moved through the bus in a burst–mode; and (2) *Non-Blocking Mode*, where the master reads or writes a single data word.

4.2. Model Checking

For model checking we considers, for both models, a set of properties describing all the possible scenarios of transactions over the bus (reading, writing, arbitration, etc.). The machine time (user time) needed for verifying the properties depends on the complexity of the original model as well as the property parameters. The time CPU required for the model checking of the properties of the PCI bus for different numbers of masters and slaves is given in Table 1. We note that the numbers of states and transitions increase exponentially as a function of the number of masters and slaves connected to the bus which explains the need for a sharp definition of the exploration domains and active actions. Similar results for the generic Master/Slave case study are given in Table 2, where, for Masters, "B" refers to Blocking and "NB" to Non-Blocking.

Number of			Model Checking			Simula-
Sla.	Mast.		CPU	Num. of FSM		-tion
	B	NB	Time(s)	Nod.	Tran.	δ(ns)
2	1	1	3.54	14	22	27.04
2	3	3	142.32	146	531	31.44
2	3	4	402.32	276	1174	33.02
2	4	4	1192.57	530	2584	35.41
3	1	1	4.32	15	27	28.01
3	3	3	186.64	147	723	36.85
3	3	4	518.73	278	1622	38.82
3	4	4	1541.32	535	3606	40.08
4	1	1	5.21	17	31	29.92
4	3	3	214.46	148	915	39.41
4	3	4	630.48	280	2070	41.11
4	4	4	2002.54	538	4630	43.25

Table 2. Master/Slave Bus: Model Checking and Simulation Results.

4.3. Assertion Based Verification

The last column in Table 1 shows a simulation evaluation of the PCI bus when implemented in SystemC including the assertions monitors. We display the average execution time per clock cycle (called δ and given in ns) as a function of the number of masters and slaves connected to the bus. This shows a very short time (few seconds) to simulate million of cycles which offers good coverage for the assertions. In this case also, we obtained similar results for the generic Master/Slave model as shown in the last column of the Table 2.

5. Conclusion and Related Work

In this paper, we presented a methodology to design and verify SystemC transactional models starting from a UML system specification and integrating an intermediate ASM layer. We proposed to upgrade the UML sequence diagram in order to capture transaction related system properties. Then both of the design at its properties are modeled in ASM to enable performing model checking. On the other hand, to cover for the state explosion problem that may result due to the system's complexity, we completed our approach by offering a methodology to apply assertion based verification re-using the already defined PSL properties. To do so, we defined a set of translation rules to transform the design's model in ASM to its implementation in SystemC. Experimental results showed good model checking results even for complex systems such as the PCI bus standard. The final SystemC models also were running a quite fast simulation enabling to offer better coverage for the whole system state space.

In [4], Gordon used the semi–formal semantics in the PSL/Sugar documentation to create a deep embedding of the whole language in the HOL theorem prover [5]. The author described how to 'execute' the formal semantics of PSL using HOL to see if it is feasible to implement useful tools that work directly from the formal semantics by mechanized proof. However, he did not provide any framework for the verification of PSL for any implementation language. Besides, they do not offer any approach to re-use the PSL properties as assertion (a very important feature in PSL).

Gawanmeh et. al. [3] completed the work of Müller *et. al.* [9] to define the semantics of SystemC 2.0 using ASM. However, that semantics could not be used directly to generate the finite state of the system because it does not satisfy to the rules we defined in Section 2.2.1. Besides, embedding the complete SystemC simulator semantics is ASM represent a huge overhead to the FSM generation.

Several proposals for system level design, in particular [11] and [12], used a combination of UML and SystemC for SoC design in general (TLM in particular). We are not aware of any other work that considered ASM as an intermediate layer between UML and SystemC to enable model checking. Besides, we focused into extracting and defining the system properties at the early design stages (from the UML sequence diagram) which makes our work complementary to existent approaches by offering and in-design verification solution.

References

[1] Accellera Organization. Accellera Property Specification Language reference manual, Version 1.01., 2004.

[2] R. Damasevicius and V. Stuikys. Application of uml for hardware design based on design process model. In *Proceedings of the 2004 conference on Asia South Pacific design automation*, pages 244–249, 2004.

[3] A. Gawanmeh, A. Habibi, and S. Tahar. Enabling SystemC Verification using Abstract State Machines. In *Forum on Specification and Design Languages*, pages 306–421, Lille, France, 2004.

[4] M. Gordon. Validating the psl/sugar semantics using automated reasoning. In *Formal Aspects of Computing*, pages 306–421, 2003.

[5] M. Gordon and T. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge Univ. Press, 1993.

[6] PCI Special Interest Group. http://www.pcisig.com/, 2004.

[7] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In *Specification and Validation Methods*, pages 9–36. 1995.

[8] Microsoft Corporation. AsmL for Microsoft .NET Framework, Microsoft., 2004.

[9] W. Müller, J. Ruf, and W. Rosenstiel. *SystemC Methodologies and Applications*. Kluwer Academic Pub., 2003.

[10] Open SystemC Initiative. www.systemc.org, 2004.

[11] C. Schulz-Key, M. Winterholer, T. Schweizer, T. Kuhn, and W. Rosenstiel. Object-oriented modeling and synthesis of SystemC specifications. In *Proc. Asia South Pacific Design Automation*, pages 238–243, 2004.

[12] V. Sinha, F. Doucet, C. Siska, R. Gupta, S. Liao, and A. Ghosh. Yaml: a tool for hardware design visualization and capture. In *Proc. Symposium on System Synthesis*, pages 9–14, 2000.