Embedding and Verification of PSL using AsmL

Amjad Gawanmeh, Ali Habibi, and Sofiène Tahar

Concordia University, Montreal, Quebec, H3G 1M8 Canada
Email:{amjad,habibi,tahar}@ece.concordia.ca

Abstract. In this paper, we propose a methodology to integrate the Property Specification Language (PSL) in the verification process of systems designed using Abstract States Machines (ASMs). We provide a complete embedding of PSL in the ASM language AsmL, which allows us to integrate PSL properties as part of the design. For the verification, we propose a technique based on the AsmL tool that translates the code containing both the design and the properties into a finite state machine (FSM) representation. We use the generated FSM to run model checking on an external tool, here SMV. Our approach takes advantage of the AsmL language capabilities to model designs at the system level as well as from the power of the AsmL tool in generating both a C# code and an FSM representation from an AsmL model. We applied our approach on SystemC designs, which are translated into AsmL models. Experimental results on a bus structure case study provided in the SystemC library showed a superiority of our approach to conventional verification.

1 Introduction and Motivation

Abstract State Machines (ASM) [2, 10] is a formal specification method for software and hardware systems that has become successful for specifying and verifying complex systems. The ASM formalism is used as a modeling language in a variety of domains both in academic and industry contexts [12]. The ASM methodology is mathematically precise, yet general enough to be applicable to a wide variety of problem areas. The ASM thesis asserts that any computing system can be described at its natural level of abstraction by an appropriate ASM. ASMs provide features to capture the behavioral semantics of programming and modeling languages, as a wide range of these languages were defined with this notion [12]. There are many languages that have been developed for ASMs, the recent one is AsmL [16], which was developed at Microsoft. We chose this language, as a common level of abstraction, to define an abstract simulator, and then model designs and properties. AsmL [6, 16] is integrated with Microsoft's software development environment including Visual Studio, MS Word, and Component Object Model (COM), where it can be compiled and connected to the .NET framework. AsmL effectively supports specification and rapid prototyping of different kinds of models. The AsmL tester can also be used to generate finite state machines (FSM) or test cases.

The Accellera Property Specification Language (PSL) [18] was developed to address the lack of information about properties and design characteristics of register transfer level (RTL) models. It provides means of specifying design properties using a concise syntax with clearly defined formal semantics. PSL permits the specification of a large class of design properties at four layers: Boolean, temporal, verification and modeling layers. It is intended to be used for functional specification to capture requirements regarding the overall behavior of a design in one hand, and as an input to verification tools using simulation or formal verification on the other hand.

In this work we propose an embedding of PSL in AsmL with an approach to verify properties on design implementations. We embed PSL properties in AsmL, to be able to reason about the behavior of the design, and its correctness against its specification. We use the AsmL tool in order to generate an FSM of the design model (including the properties). This approach enabled the verification of PSL properties on designs using classical model checking tools, for example SMV [15]. For this, we translate the generated FSM into the input language of the SMV tool.

Figure 1 shows a sketch of our methodology for the embedding and verification of PSL in AsmL, where the contributions of this paper are indicated by a dashed box. We applied our approach to designs modeled in SystemC [22], which are translated into AsmL [4, 11]. We used the bus structure of the SystemC library to show the feasibility and performance of our approach. The experimental results proved the practicality of our methodology as a solution to the verification problem of SystemC designs.

The rest of the paper is organized as follows: Section 2 presents related work to ours. Section 3 describes our embedding for PSL in AsmL. In Section 4 we provide an application for our approach to SystemC verification with a case study to illustrate our methodology. Finally, Section 5 concludes the paper and points to a few future work directions.

2 Related Work

In [9], Gordon used the semi–formal semantics in the PSL/Sugar documentation [18] to create a deep embedding of the whole language in the HOL theorem prover [8]. The author developed the formal definition of the full PSL language in HOL. The combination of PSL/Sugar and higher–order logic is quite expressive and provides temporal logic constructs as higher level syntactic sugar for higher order–logic, thereby enabling properties to be formulated elegantly. Gordon *et al.* [7] described how to 'execute' the formal semantics of PSL using HOL and investigated the feasibility of implementing useful tools to conduct automatic verification of PSL from the formal semantics. They implemented two experimental tools: an interpreter that evaluates whether a finite trace, satisfies a PSL formula, and a compiler that converts PSL formulas to checkers in an intermediate format suitable for translation to HDL to be included in simulation test–benches. However, they did not provide any framework for the verification of PSL for any implementation language.

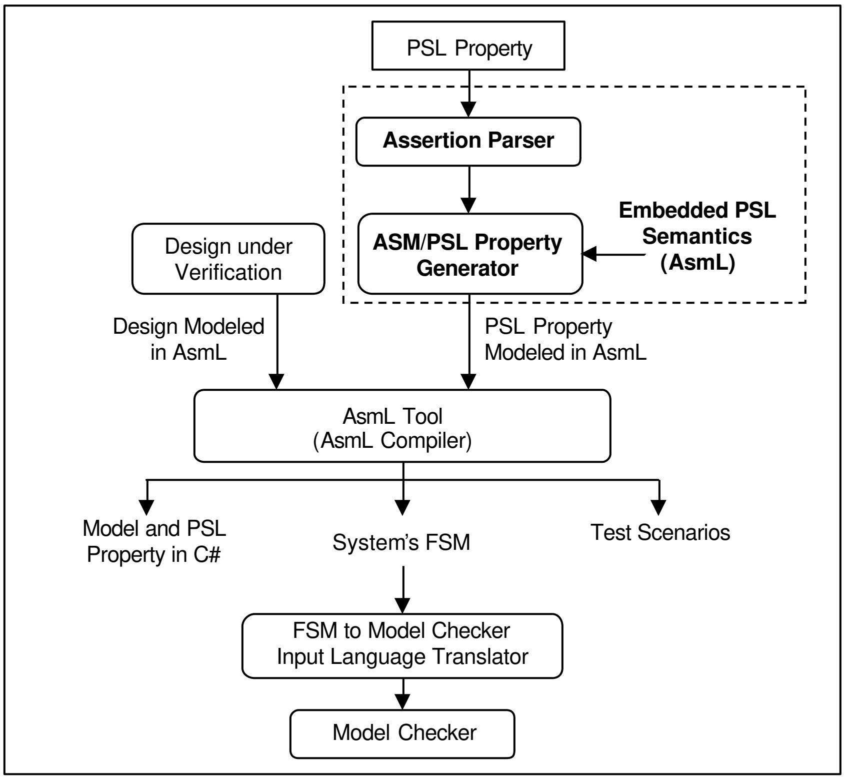

Fig. 1. Methodology for verification of PSL in AsmL.

In a similar work, Claessen and Martensson [3] defined an operational semantics for a weak fragment of PSL, mainly the safety property subset of PSL, and then proved the correctness of these semantics with respect to a denotational semantics. They do not provide definitions for all PSL operators like clock operators and sequential composition, and yet, there is no execution for these semantics that provides verification solution.

There has been a potential work on ASM verification as discussed by Börger and Stärk [2]. Applying model checking algorithms on ASM was introduced in [24], where transformation algorithms are provided to transform ASM models into different verification tools. Two approaches were adopted: the first provides a transformation to the language of a symbolic model checker, SMV [24], and the second to the MDG verification tool [5]. Spielmann [19] investigated the problem of verifying a class of restricted abstract state machine programs automatically. The work we present here, is different since it provides a solution for the verification problem of system level design languages based on semantics definitions and executions of PSL and SystemC.

Stärk *et al.* [20] used an ASM-based modularization technique to define a structured sequence of mathematical models for the statics and dynamics of the Java programming language and for the Java Virtual Machine (JVM), covering the compilation of Java programs to JVM code and the JVM bytecode verifier. They present proofs of correctness, completeness, and type safety for the language and the Java machine. Börger *et al.* [1] used the method developed in [20] to define the semantics of C# programs in ASM, which provided a simple way to reflect those run-time-related features encountered upon executing a given C# program and allowed specifying the static and dynamic parts of the semantics separately. The dynamic semantics of the language is captured operationally by ASM rules which describe the run-time effect of program execution on the abstract state of the program, the static semantics comes as a declarative description of the relevant syntactical and compile-time checked language features. In a complementary work, Stärk and Börger [21] extended the modular definition of the semantics of C# in [1] by a new module for multi-threaded C# focusing on purely managed, fully portable threading features of C# and the .NET common language runtime. Jula and Fruja [14] provided an executable AsmL semantics for these C# semantics. In a later work, Jula [13] extended the work in [1] to handle C# 2.0 specific features like generics, anonymous methods and iterator blocks.

ASM has been used thoroughly to define the operational semantics of programming languages like C++, Prolog, SDL, and Standard ML [12]. However, these semantics definitions provide no execution of the language semantics itself in order to give a solution to design problems like verification.

3 Embedding PSL in AsmL

PSL is an implementation independent language to define properties (also called assertions). It does not replace, but complements existing verification methodologies like VHDL and Verilog test benches. PSL presents a different view of the design and may imply FSMs in the implementation. The syntax of PSL is very declarative and structural which leads to sustainable verification environments. Both VHDL and Verilog flavors are provided. PSL consists of four layers based on the functionality:

The Boolean layer to build expressions which are used in other layers, specifically the temporal layer. Boolean expressions are evaluated in a single evaluation cycle.

The temporal layer is used to describe properties of the design, in addition to simple properties, this layer can describe properties that involve complex temporal relations. Temporal expressions are evaluated over a series of evaluation cycles.

The verification layer is used to tell the verification tool what to do with the properties described by the temporal layer.

The modeling layer is used to model behavior of design inputs for formal verification tools, and to model auxiliary parts of the design that are needed for verification.

This layered approach allows the expressing of complex properties from simple primitives. A property is built from three types of building blocks: Boolean expressions, sequences, which are themselves built from Boolean expressions, and finally subordinate properties. Sequences, referred to as SEREs (Sequential Extended Regular Expressions), are used to describe a single– or multi–cycle behavior built from Boolean expressions.

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 AsmL, at lower design levels since the AsmL tool can automatically compile them into a C# or .NET code. This latter code can be compiled and executed with the concrete SystemC level.

3.1 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 AsmL 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 [18], and the formal semantics definition in HOL [9].

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 AsmL's *AsmL.Boolean* and *AsmL.String*, respectively. The *PSLBit* type is constructed using the enumerated structure One, Zero, X, and Z. The *PSLBitVect* type extends the *PSLBit* type and offers additional operations such as access to the bit vector contents. Finally, the *PSLNumeric* type extends the AsmL *Integer* type (AsmL.Integer) by adding some conversion methods from *PSLBitVector* to integers and vice–versa.
2. *PSL Expressions* construct properties using the implication and equivalence operators. Both operators are built using AsmL's *implies* operator.
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()*). For both methods, we define a queue structure that extends the *PrimitiveArray* class of AsmL, to store the values of the signals (*PSL_Bit_Vector_Queue* for the *PSLBitVector* type). We note that all the methods over the Boolean layer are overridable according to the type of the input. This approach simplifies writing the properties in AsmL syntax as they will look very close to the PSL structure.

Figure 2 shows the AsmL code for PSL_Bit_Vector class with the method $IsInitialized()$ that checks if a BitVector is initialized.

```
class PSL_BitVector
  var m_size as Integer = 1
  var m_sum  as Integer = 0
  var m_array as PrimitiveArray of PSL_Bit = null
  public IsInitialized() as Boolean
    non_initailized = (exists x in {1..m_size} where
        (m_array(x).m_value = X or m_array(x).m_value = Z))
    return not non_initailized
```

Fig. 2. AsmL Embedding of PSL BitVector.

3.2 Temporal Layer

The most important part of this layer is the Sequential Extended Regular Expressions (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. *PSL_Sequence* extends the *PSL_SERE* class. It adds operations needed to create and update the SERE.
2. *Properties* in the form of 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.

3.3 Verification Layer

This layer is intended to tell the verification tool how to perform the verification process. It allows the construction of 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 temporal layer class *PSL_Property* defined above.
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.

```
class PSL_SERE
  var m_size as Integer = 0
  var m_seq as Seq of Boolean
  var m_actualState as Integer = 0
  var m_evaluation as SERE_Evaluation = NOT_STARTED
  var m_evaluationState as SERE_Evaluation = NOT_STARTED
  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
```

Fig. 3. AsmL Embedding of PSL SERE.

Figure 4 shows the example of the *PSL_VerificationLayerUnit.CopyFrom()* and *PSL_VerificationLayerUnit.CopyTo()* methods. These latter are usually used to construct the unit by copying properties from or into other existent units, respectively.

```
class PSL_VerificationLayerUnit
  var m_name as String = ""
  var m_size as Integer = 0
  var S as Seq of PSL_FL_Property = null
  CopyFrom(vunit as PSL_VerificationLayerUnit)
    forall i in {1..m_size}
      me.AddProperty(vunit.S(i))
  CopyTo(vunit as PSL_VerificationLayerUnit)
    forall i in {1..m_size}
      vunit.AddProperty(me.S(i))
```

Fig. 4. Embedding PSL Verification layer in AsmL.

3.4 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 current embedding.

PSL properties are embedded in AsmL as assertions, the assertion here means the validity of the property and 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 that 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 assertion will be embedded in every state in the FSM generated by the AsmL tool as a Boolean states variable, and therefore the FSM will include, by construction, a Boolean state variable giving the state of the property. Model checking tools, like SMV, can be used to check the correctness of the property on the generated FSM. The fact that we embed the property in the generated FSM provides a clear definition of the property, since its state variables are evaluated in every state by the AsmL tool, and the model checker just needs to perform reachability analysis on the model without the need to calculate the valuation of the property in every state. The final generated FSM is concrete and includes only Boolean variables to represent the state of the PSL properties. This will significantly reduce the verification time as will be illustrated in the case study in the next Section.

4 Application with SystemC

In this section, we show an application to SystemC verification based on the proposed PSL embedding in AsmL. SystemC [22] was introduced as a new system level language to overcome the problem of the growth in complexity and size of systems combining different types of components, including microprocessors, DSPs, memories, embedded software, etc. SystemC meets the need for a system level language that can fill the gap between hardware description languages (HDLs) and traditional software programming languages. SystemC comprises C++ class libraries and a simulation kernel used for creating behavioral and register transfer level (RTL) designs. It can provide the common development environment needed to support software engineers working in C/C++ and hardware engineers working in HDLs such as Verilog or VHDL.

In order to apply PSL property checking on a design language, here SystemC, that language should also be embedded in AsmL. In fact, the semantics of SystemC 2.0 using ASM were first defined by Müller *et. al.* [17]. However, fundamental features of SystemC 2.0 were not defined, including some SystemC primitive and hierarchical channels, design rules for SystemC channels, and most importantly the semantics of the SystemC simulator. An extended semantics of the SystemC simulator has been defined using ASM rules in [4] including a model for the SystemC simulator, the primary classes that are necessary to model SystemC designs and execute them, and the semantics of the main functions of the following components: SystemC signal, SystemC FIFO, SystemC mutual exclusive (MUTEX) channel, message queue, request–grant protocol, FIFO with handshake protocol [4].

The above embedding allowed us to translate SystemC designs into AsmL models. Together with the PSL embedding described in this paper, we can generate a single model combining both the properties and the design model in one single formalism, namely AsmL. Details about the SystemC to AsmL and AsmL to SystemC transformations can be found in [11].

In order to illustrate the proposed verification methodology, we consider an extension of the Simple Bus Prototype Structure provided as part of the SystemC distribution [22]. It includes several SystemC components and shows the principles of using SystemC at the transactional level. Besides, some of the sample properties we checked, including liveness and safety, cannot be verified using simulation. They require the usage of formal techniques such as model checking.

4.1 Bus Description

Figure 5 shows the bus structure with N masters and M slaves. Each master is identified by a unique priority, that is represented by an unsigned integer number. The lower this priority number is, the more important the master is. Each master communicates with the bus via an interface, which describes the communication between masters and the bus. The slave interface describes the communication between the bus and the slaves. Multiple slaves can be connected to the bus. The slave returns an acknowledgment immediately upon receiving the data. The arbiter is responsible for choosing the appropriate master when there is more than one connected to the bus. It performs the selection according to the priority of the master. There are two possible modes for the bus: (1) *Blocking Mode*, where data is moved through the bus in a burst–mode. Here, the transaction cannot be interrupted by a request with a higher priority, (2) *Non-Blocking Mode*, where the master reads or writes a single data word. Figure 6 shows the protocol used in both modes of operation.

4.2 Bus Properties

For illustration purposes, we considered two properties for the bus architecture: one for the non–blocking master mode, and the other for the blocking master mode.

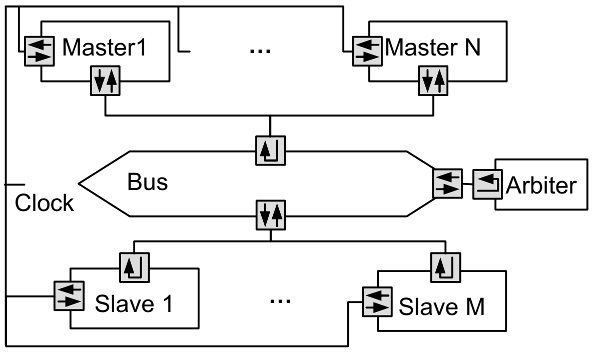

Fig. 5. Simple bus structure.

Property P1:

If ((*MasterBlock.Request* = *true*) & (*BusStatus* = *OK*) &
(*MasterBlock.Priority is the highest*)) ***then***
(*MasterBlock*[3] = *OKSend*) &
(*BusStatus*[3] = *Used*) &
(*MasterBlock.DestSlave*[4] = *Recev*) &
(*MasterBlock.DestSlave*[5] = *Ack*) &
(*MasterBlock*[7] = *Done*) &
(*Bus*[8] = *Ready*)

meaning that when a blocking master generates a request while it has the highest priority to use the bus and the bus is available, then at the third clock cycle, the status of the master should be $OKSend$, and the bus should be in the $Used$ status. Then the status of the destination slave should be *Recev* at clock cycle 4, and *Ack* at clock cycle 5. The status of the master should be *Done* at clock cycle 7, and finally the bus becomes ready to handle new requests (i.e., bus status set to *Ready*) at clock cycle 8. This property is illustrated as a sequence diagram in Figure 6(a).

Property P2:

If ((*MasterNBlock.Request* = *true*) & (*BusStatus* = *OK*)
& (*MasterNBlock.Priority is the highest*)) ***then***
(*MasterNBlock*[3] = *OKSend*) &
(*BusStatus*[3] = *Used*) &
(*MasterNBlock.DestSlave*[4] = *Recev*) &
(*MasterBlock*[5] = *Done*) &
(*Bus*[5] = *Ready*)

This property can be interpreted in a similar way as property $P1$ and is illustrated in Figure 6(b).

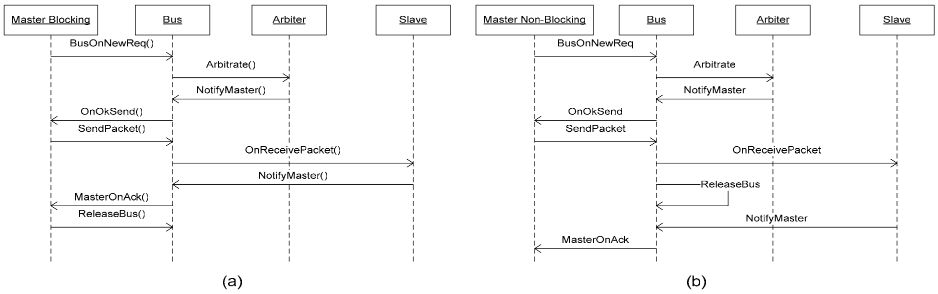

Fig. 6. Simple bus modes: (a) Blocking mode (b) Non-blocking mode.

Both properties were defined in AsmL based on the embedding of PSL layers. Figure 7 shows the definition of $P1$ as an example. The property is included in a PSL unit as an implication of two sequences $seq1$ and $seq2$, which are formed from basic Boolean items ($Bi1()$ and $Bi2()$ for $seq1$ and $Bi3()$ through $Bi7()$ for $seq2$). The construction of the above unit includes four steps:

- Creating the basic Boolean items: $Bi1()$ to $Bi7()$.
- Creating the sequences: $seq1$ and $seq2$.
- Constructing the implication property ($P1$) from $seq1$ and $seq2$ using the implication operator.
- Putting $P1$ into an embedded PSL unit.

4.3 Experimental Results

For experimental purposes, we generated the FSM for different combinations of the number of masters and slaves in order to evaluate the generation algorithm used inside the AsmL tool, and to illustrate the efficiency of our methodology[1]. In every state, we include the evaluation of the property $P1$ that will be used by the model checker. $P1$ can take any of these five states: *NotStarted* (the required conditions to evaluate the property are not satisfied yet), *Started* (all the required conditions to start evaluating the property are satisfied), *Evaluating* (the evaluation of the property is in progress and no error is found), *FALSE* (the property is found to be false in this state) and *TRUE* (the property is true at this state). The property in SMV is defined as:
$Property1:$
$assert(not(P1\_Status = FALSE)\ and\ (P1\_Status = Started) \Rightarrow F(P1\_Status = TRUE)).$

The machine time (user time) needed for generating the state machine depends on the complexity of the original model as well as the size of the generated

[1] The SystemC code, AsmL code, AsmL configuration files, and the generated FSMs for the case study are available at http://hvg.ece.concordia.ca/Research/SoC/ASM.

```
//Blocking Masters Instances
var masterB1 as C_MasterBlocking = new C_MasterBlocking
var masterB2 as C_MasterBlocking = new C_MasterBlocking
MASTERSB = {masterB1, masterB2}

Bi1() as Boolean Bi: Boolean Item
  return exists master in MASTERSB where master.m_status = MasterReq
Bi2() as Boolean
  return exists master in MASTERSB where
         master.m_Priority = max y | y in {MasterReq.m_Priority}
Bi3() as Boolean
  return master.m_status = MasterOKSend
Bi4() as Boolean
  return exists slave in SLAVES where
         slave.ID = master.m_Destination and
         slave.m_Status = SlaveReceive
Bi5() as Boolean
  return exists slave in SLAVES where
         slave.ID = master.m_Destination and
         slave.m_Status = SlaveAck
Bi6() as Boolean
  return master.m_status = MasterReady
Bi7() as Boolean
  return bus.m_status = BusOK

var seq1 as PSL_SERE = PSL_SERE(2)
         seq1.AddElement(param1, param2)
var seq2 as PSL_SERE = PSL_SERE(5)
         seq2.AddElement(Bi3, Bi3, Bi4, Bi5, Bi6, Bi7)
var property as PSL_FL_Property = PSL_FL_Property()
property.AddImplication(seq1,seq2) var Assertion1 as
PSL_VerificationLayerUnit = new
PSL_VerificationLayerUnit(''Assertion1'')
Assertion1.AddProperty(property)
```

Fig. 7. Definition of the PSL Property P1 in AsmL.

state machine. Figure 8 displays the user time and number of transition for different combinations of the number of slaves and masters operating on the bus for generating the FSM model and PSL property. For example, for four slaves and eight masters, there were 4630 transitions, and it took around half an hour to generate the state machine. The experiments were conducted on a Pentium IV processor (2.4 GHz) with 256 MB of memory.

The above experimental results show that the number of transitions and machine time grow exponentially with the number of operating masters. Therefore, the AsmL tool could not handle more complex combinations like seven masters

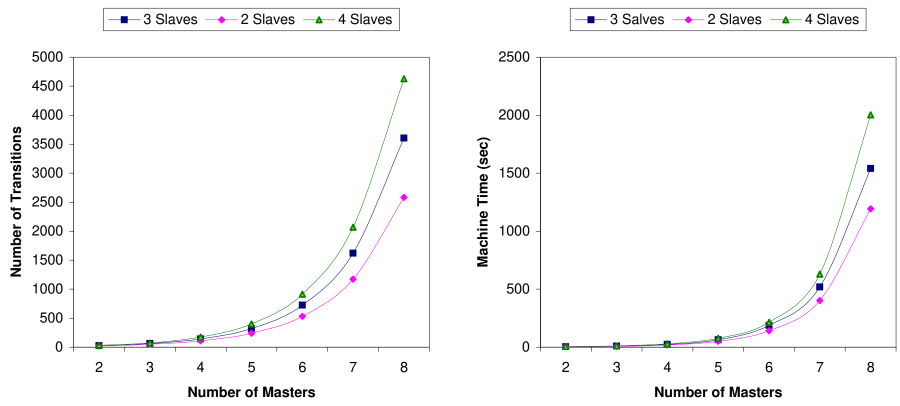

Fig. 8. FSM generation results for the bus model and the PSL properties.

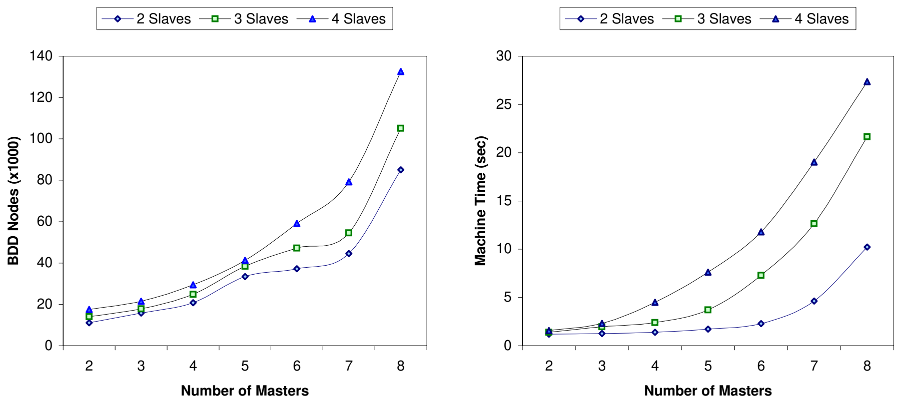

Fig. 9. SMV model checking results for the bus model and the PSL properties.

with four slaves and so failed to generate the state machine. This is because of the limitations of the algorithm used inside the AsmL tool.

The AsmL tool provides the option to generate the FSM in the *dot* format. We translated this representation into SMV code and conducted model checking using the SMV tool. Figure 9 provides the verification results of the above properties for the same set of combinations of masters and slaves. Accordingly, the PSL properties were found correct in less than 30 sec., even for the most complex cases. Therefore, we believe any improvement in the approach will come at the level of improving the generation of the FSM from the AsmL code. In this paper, we used the algorithm implemented in the AsmL tool as a black–box.

We will consider in a future work the introduction of new algorithms that will take advantage of both the SystemC system level design particularities and the property content and structure as defined in PSL.

5 Conclusion

In this paper, we presented an embedding of the Property Specification Language (PSL) in Abstract State Machines Language (AsmL). We provided a deep embedding in AsmL of the three hierarchical layers: Boolean, temporal, and verification of PSL. An AsmL model combining both the reduced design and the PSL property is input to the AsmL tool, which compiles it into C#, and generates its FSM. The verification of PSL properties is performed using the SMV model checker after translating the FSM model (given in *dot* format) into the input language of SMV.

We used a previously defined embedding of the semantics of SystemC components library and simulator in order to apply PSL properties verification on SystemC designs. We illustrated this approach through a case study of the bus structure model, on which we verified several PSL sample properties for a system including up to eight masters and four slaves. This was not possible by directly verifying the PSL properties on the original SystemC design model using any of the publicly available tools. As future work, we plan to improve the proposed algorithm to generate more efficient FSMs that take into consideration the property under verification. We also intend to test the approach with a larger case study and apply verification techniques such as assertion based verification and test case generation. Furthermore, we intend to evaluate the verification of PSL assertions by simulation using the interface offered by the AsmL tool to the .NET framework. It is also possible to formalize the PSL into ASM (not AsmL) in order to provide rigorous definitions for PSL semantics. We also propose to explore the possibility of applying this approach on other system modeling languages like SystemVerilog [23].

References

1. E. Börger, G. Fruja, V. Gervasi, and R. Stärk. A high-level modular definition of the semantics of C#. Theoretical Computer Science, 2004
2. E. Börger and R. Stärk. Abstract State Machines: A Method for High-Level System Design and Analysis. Springer–Verlag, 2003.
3. K. Claessen and J. Martensson. An Operational Semantics for Weak PSL. In A. Hu and A. Martin (eds.), Formal Methods in Computer-Aided Design, LNCS 3312, Springer–Verlag, pp. 337–351. November 2004.
4. A. Gawanmeh, A. Habibi and S. Tahar: Enabling SystemC Verification using Abstract State Machines, Proc. Languages for Formal Specification and Verification, Forum on Specification & Design Languages, September 2004.
5. A. Gawanmeh, S. Tahar and Kirsten Winter. Formal Verification of ASM Designs using the MDG Tool. In A. Cerone, P. Lindsay (eds.), Software Engineering and Formal Methods, IEEE Computer Society, pp. 210–219. September 2003.

6. W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating Finite State Machines from Abstract State Machines. Software Engineering Notes, 27(4):112–122, 2002.
7. M. Gordon, J. Hurd and K. Slind. Executing the Formal Semantics of the Accellera Property Specification Language by Mechanised Theorem Proving. In D. Geist and E. Tronci (eds.), Correct Hardware Design and Verification Methods, LNCS 2860, Springer–Verlag, pp. 200–215, October 2003.
8. M. Gordon and T. Melham. Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic, Cambridge, U.K., Cambridge Univ. Press, 1993.
9. M. Gordon. Validating the PSL/Sugar Semantics Using Automated Reasoning. Formal Aspects of Computing, 15(4): 406–421, 2003.
10. Y. Gurevich. Evolving Algebras 1995: Lipari Guide. In E. Börger (ed.), Specification and Validation Methods, Oxford University Press, 1995.
11. A. Habibi and S. Tahar. On the Transformation of SystemC to AsmL using Abstract Interpretation. In Proc. Int. Workshop on Abstract Interpretation for Object Oriented Languages. Paris, France, January 2005.
12. J. Huggins. Abstract State Machines website. http://www.eecs.umich.edu/gasm, 2003.
13. H. Jula. ASM semantics for C# 2.0. In Proc. Abstract State Machines 2005, Université de Paris 12, France, March 2005.
14. H. Jula and N. Fruja. An executable specification of C#. In Proc. Abstract State Machines 2005, Université de Paris 12, France, March 2005.
15. M.L. McMillan. Symbolic Model Checking, Kluwer Academic Pub., 1993.
16. AsmL for Microsoft .NET (version 2.1.5.7), Microsoft. http://www.research.microsoft.com/foundations/asml, 2003.
17. W. Müller, J. Ruf, and W. Rosenstiel. SystemC Methodologies and Applications. Kluwer Academic Pub., 2003.
18. Accellera Property Specification Language Reference Manual, Version 1.01. http://www.accellera.org, 2004.
19. M. Spielmann. Automatic Verification of Abstract State Machines. In N. Halbwachs and D. Peled (eds.), Computer Aided Verification, LNCS 1633, Springer–Verlag, pp. 431–442, 1999.
20. R. Stärk, J. Schmid, and E. Börger. Java and the Java Virtual Machine Definition, Verification, Validation. Springer–Verlag, 2001.
21. R. Stärk and E. Börger. An ASM Specification of C# Threads and the .NET Memory Model. In W. Zimmermann and B. Thalheim (eds.), Abstract State Machines. Advances in Theory and Practice, LNCS 3052, Springer–Verlag, pp. 38–60, 2004.
22. SystemC. http://www.systemc.org, 2004.
23. SystemVerilog. http://www.systemverilog.org, 2004.
24. K. Winter. Model Checking Abstract State Machines, Ph.D. thesis, Technical University of Berlin 2001.