Journal of Circuits, Systems, and Computers

A Design for Verification Approach using an Embedding of PSL in AsmL

Amjad Gawanmeh and Sofiène Tahar
Concordia University,
Montreal, Quebec, H3G 1M8 Canada
{*amjad,tahar*}*@ece.concordia.ca*

Haja Moinudeen *
Poseidon Design Systems,
Bangalore, India
haja@poseidon-systems.com

Ali Habibi *
MIPS Technologies, Inc.
Mountain View, California, 94043 USA
habibi@mips.com

Received ( )
Revised ( )
Accepted ( )

In this paper, we propose to integrate an embedding of Property Specification Language (PSL) in Abstract State Machines Language (AsmL) with a top-down design for verification approach in order to enable the model checking of large systems at early stages of the design process. 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 C# code and FSMs from AsmL models. We applied our approach on the PCI-X bus standard, which AsmL model was constructed from the informal standard specifications and a subsequent UML model. Experimental results on the PCI-X bus case study showed a superiority of our approach to conventional verification.

Keywords: PSL; Abstract State Machines; AsmL; Model Checking; PCI-X Bus.

*This work was done while this author was working at Concordia University.

1. Introduction and Motivation

With the advent of high technology applications, an increasingly evident need has been that of incorporating the traditional microprocessor, memories and peripherals on a single silicon. System level modeling is used 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. System level languages can fill the gap between hardware description languages (HDLs) and traditional software programming languages. Therefore, the modeling and verification process of systems level designs, at early stage of the design process, is very challenging.

The verification of systems at early stages of the design process is a serious bottleneck in the system design flow. While simulation is the most widely used verification technique, it is unable to guarantee the correctness of the design with respect to its specification. On the other hand, model checking is considered as a relevant technique to cover for simulation insufficiencies. Nevertheless, direct model checking may not be feasible due to the complexity of the designs. Besides, the state explosion problem led, for complex systems, to the use of assertion based verification (ABV) where the property under verification is turned into a monitor, checked by simulation and evaluated using coverage metrics. Therefore, there is a need for verification solutions for industrial size designs at the system level.

Abstract State Machines (ASM) [1,14] 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 [18]. 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 [18]. There are many languages that have been developed for ASMs, the recent one is AsmL [15], which was developed at Microsoft Research. We chose this language, as a common level of abstraction, to define an abstract simulator, and then model designs and properties. AsmL 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 (Asmlt) can also be used to generate finite state machines (FSM) or test cases [9].

The Accellera Property Specification Language (PSL) [23] 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, verifica-

tion and modeling layers. PSL 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 the work proposed in [7] we used a bottom-up approach in order to accomplish verification for SystemC designs based on embedding PSL in AsmL, and using our ASM semantics for SystemC [6,17]. We embedded PSL properties in AsmL, to be able to reason about the behavior of the design, and its correctness against its specification. Then, we used 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 instance SMV [19]. For this, we translate the generated FSM into the input language of the SMV tool.

In this paper, however, we use a top-down approach for verification of systems at first stages of the design process based on the design for verification approach proposed in [16] and [20]. Here we start with an informal specification of system and model it with the Unified Modeling Language (UML) in order to have a clear view of the design modules and their interactions. Then, we construct an Abstract Machine Language (AsmL) model from the UML representation. This paper integrates this design for verification approach into the verification methodology presented in [7] and applies the methodology on an industrial size case study of the PCI-X bus standard [22]. We used the PCI-X bus 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 system level designs.

The rest of the paper is organized as follows: Section 2 presents related work to ours. In Section 3 we provide the integration of the design for verification approach into the PSL-AsmL verification methodology. Section 4 describes our embedding for PSL in AsmL. Section 5 illustrated our approach on the case study of the PCI-X bus. Finally, Section 6 concludes the paper and points to a few future work directions.

2. Related Work

In [13], Gordon used the semi–formal semantics in the PSL/Sugar documentation [23] to create a deep embedding of the whole language in the HOL theorem prover [12]. 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.* [11] 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.

In a similar work, Claessen and Martensson [2] 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 [1]. Applying model checking algorithms on ASM was introduced in [28], 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 [28], and the second to the MDG verification tool [8]. 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 in AsmL.

Wallace [26] used the ASM notation to define operational semantics of C++ programming language focusing on its object oriented and other advanced features like classes, inheritance, operator overloading, virtual functions, templates, and exception handling. Definitions for other languages semantics such as Prolog, SDL, and Standard ML using ASM can be found in [18]. These semantics definitions provide no execution of the language semantics itself in order to give a solution to design problems like verification.

ASM has been used widely to define the operational semantics of programming languages like C++, Prolog, SDL, and Standard ML [18]. However, these semantics definitions provide no execution of the language semantics itself in order to give a solution to design problems like verification.

Other related work to ours concerns finite-state verification includes the work on Bandera project [5] interfacing Java code to model checking tools like SMV and SPIN by applying program analysis, abstraction, and transformation techniques. In its actual status, Bandera cannot handle SystemC designs because any analysis of a SystemC code must go through the whole simulation environment as well as SystemC defined data-types and classes. In [10] an approach is presented to add assertion checkers to SystemC. This previous work is different from our methodology since the properties in [10] are restricted to the notation of property checker from Infineon Technologies AG then translated to synthesizable SystemC instructions while we consider any PSL property. In his PhD thesis, Habibi [16] proposed a combination of various techniques to verify system level languages, in particular SystemC. His approach combined static code analysis, model checking and assertion based verification for system level designs. Habibi focused on the verification problem of the SystemC case, while this work extends his approach to provide a top-down design for verification approach.

There exists some other related works to ours in the context of PCI technologies design and verification environment. Shimizu *et al.* [24] presented a specification of the PCI bus as a Verilog monitor. Oumalou et al. [21] implemented the PCI bus in SystemC. In [4], a bridge for PCI Express and PCI-X was designed in Verilog at RTL and also verified and synthesized. Chang et al [3] proposed a simulation based PCI-X verification environment using C and Verilog. Wang *et al.* [27] proposed a similar verification environment as in [3] for PCI-X bus functional models using VERA. Yu *et al.* [29] extended verification environment in [3] to support PCI, PCI- X and PCI-Express in a single platform.

3. Verification Methodology

Our methodology is a top-down approach for verification of systems at first stages of the design process based on the design for verification approach proposed in [16] and [20]. Here we start with an informal specification of system and model it with the Unified Modeling Language (UML) in order to have a clear view of the design modules and their interactions, we use UML class and sequence diagrams in order to capture all the design requirements efficiently. Class diagrams are used to represent all the core components of the design and the sequence diagrams are used to model all the sequences of various transactions. The reason of this UML modeling is to represent the system graphically and the ability to represent the notion of sequences of the bus transaction using sequence diagrams. Besides, using our UML representation, designers can model the bus in any Object Oriented languages.

The next step, is to construct an Abstract Machine Language (AsmL) model from the UML representation. The translation from UML representation to AsmL is manual but straightforward. We map the UML classes into AsmL classes and the sequence diagrams are used to represent the order of method executions in AsmL. Unlike the UML model, the AsmL model can be executed and validated. The properties of the bus design are specified in PSL and embedded with the AsmL model. They are basically extracted from the sequence diagrams and encoded in the PSL syntax. Next, model checking is performed on the AsmL model using AsmL tester (Asmlt). The model checking process ends to: (1) a completion either with success or failure of the property; or (2) a state explosion. In case of failure, we correct the UML representation and redo the AsmL translation. This procedure is repeated until all the properties pass with success or do not complete.

Figure 1 shows a sketch of our methodology for the embedding and verification of PSL in AsmL. Properties are embedded in every state in the FSM generated by the Asmlt and is represented by two Boolean state variables. The first represents whether a property can be evaluated or not and the second denotes the state of the property in the current state. A correct verification process results in the generation of the system's FSM. If the property is not verified, then an error trace would be generated. In the case of state explosion, the FSM generation will become unsuccessful.

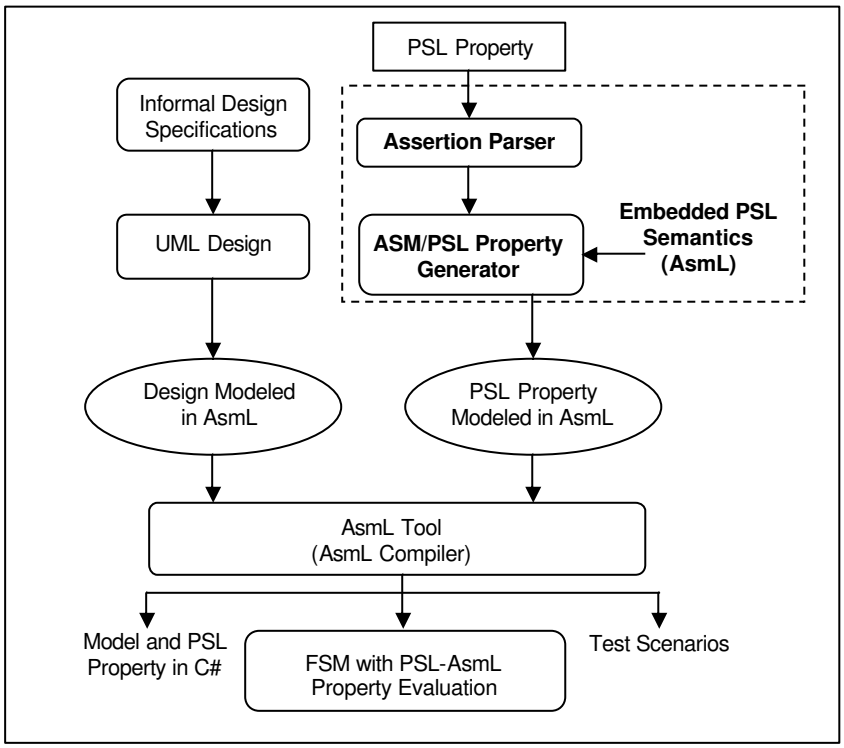

Fig. 1. Design for Verification Methodology based on Embedding PSL in AsmL.

This methodology is different from our previous work in [7] in two major directions: the considered model under verification, and the way we perform model checking. The work in [7], which is highlighted in the dashed box in Figure 1, targets low level designs in SystemC, while this work is a design for verification approach starting from informal specifications. On the other hand, the way we perform model checking here is different than the classical approach used in [7], where an external tool is used to perform model checking. In this approach, however, once the modeling in AsmL is done, we include the defined PSL properties in the AsmL design. Then, the AsmL design with the properties is fed to the Asmlt tester that generates an FSM of the model. Using Asmlt, we perform state exploration which gives the notion of model checking. We encode the properties evaluation in every state, which enables checking its correctness on-the-fly while executing the FSM generation algorithm [9] (part of the Asmlt). Any incorrect property detection halts the reachability algorithms and outputs a sub-portion from the complete FSM, which can be used to identify counter-examples.

In the next section, we show the details of embedding the layers of PSL in AsmL, then, we apply our methodology on an industrial size case study of the PCI-X bus,

where we show the efficiency of the top-down design for verification approach, and we verify several properties on the bus.

4. 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 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 C# or .NET code.

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 as described.

4.1. *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 [23], and the formal semantics definition in HOL [13].

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.

4.2. *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.

```
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.

4.3. *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.

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.

4.4. *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.

4.5. *Verifying PSL Properties*

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 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.

5. Case Study: Modeling and Verification of the PCI-X Bus

5.1. *PCI-X Bus Description*

Improvements in processors and peripheral devices have caused conventional PCI technology to become a bottleneck to performance scalability. The introduction of the PCI-X technology [22] has provided the necessary bandwidth and bus performance needed to avoid the I/O bottleneck, thus achieving optimal system performance. For instance, version 2.0 of PCI-X specifies a 64-bit connection running at speeds of 66, 133, 266, or 533 MHz, resulting in a peak bandwidth of 533, 1066, 2133 or 4266 MB/s, respectively.

PCI-X provides backward compatibility by allowing devices to operate at conventional PCI frequencies and modes. Moreover, PCI-X peripheral cards can operate in a conventional PCI slot, although only at PCI rates and may require a 3.3 V conventional PCI slot. Similarly, a PCI peripheral card with a 3.3 V or universal card edge connector can operate in a PCI-X slot, however the bus clock will remain at a frequency acceptable to the PCI card. Figure 5 shows the general architecture of PCI-X with one initiator (master) and target (slave). There is an arbiter that performs the bus arbitration among multiple initiators and targets. Unlike in conventional PCI bus, the arbiter in PCI-X monitors the bus in order to ensure good functioning of the bus.

Both PCI-X initiator and target have a pair of arbitration lines that are connected to the arbiter. When an initiator requires the bus, it asserts REQ#. If the arbiter decides to grant the bus to that initiator, it asserts GNT#. FRAME# and IRDY# are used by the arbiter to decide the granting of an initiator request for the bus. Unlike PCI, the targets can only insert wait states by delaying the assertion of TRDY#. TRDY# and IRDY# have to be asserted for a valid data transfer. An initiator can abort the transaction either before or after the completion of the data transfer by de-asserting the FRAME# signal. In contrast, a target can terminate a bus transaction by asserting STOP#. If STOP# is asserted without any data transfer, the target has issued a retry and if STOP# is asserted after one or more data phases, the target has issued a disconnect. Unlike PCI, the target has

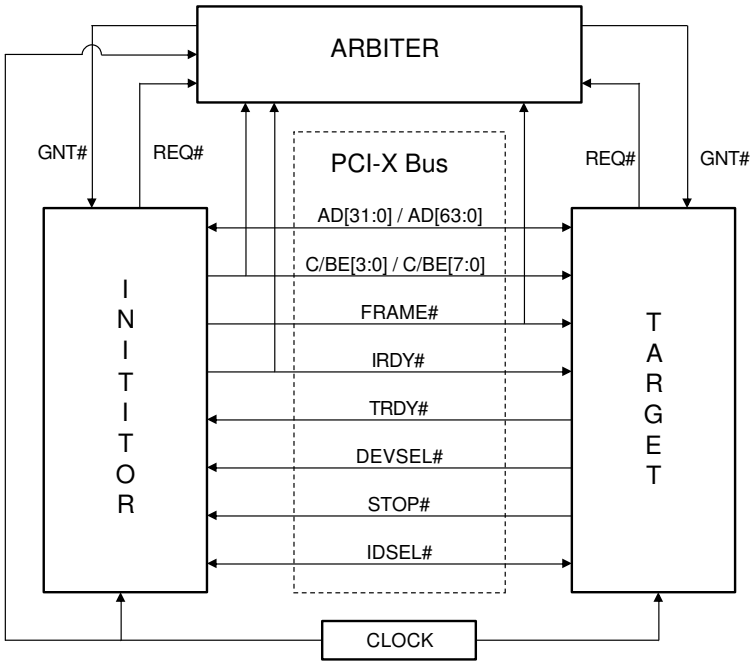

Fig. 5. General Architecture of PCI-X.

also REQ# and GNT# that are connected to the arbiter. This facilitates the split transaction of PCI-X which does not exist in conventional PCI. In split transaction, the initiators and targets switch their roles. Split transaction is very useful if a target cannot continue the current transaction. In this case, the target will memorize the transaction and signal a split telling the initiator not to retry IO read. When the data is ready, the target will send the initiator a Split Completion containing the data. The addition of PCI-X Split Completion frees up the bus for other transactions, making PCI-X more efficient than PCI. This notion of split transaction of PCI-X and its high bandwidth capacity makes the PCI-X bus pretty complex.

5.2. *UML Modeling of PCI-X Bus*

From the specification of PCI-X, we identify the core components, i.e., initiators, targets, and arbiters, which will be represented as classes, where specific instances of the components are called as objects. In addition to these four components, we also added another component, the Simulation Manager (SimManager), in order to have a notion of updates. Figure 6 shows these five classes, where each has its own data members and methods with their access types. It also shows the relationship among each classes. For instance, the relationship between the arbiter and the initiators

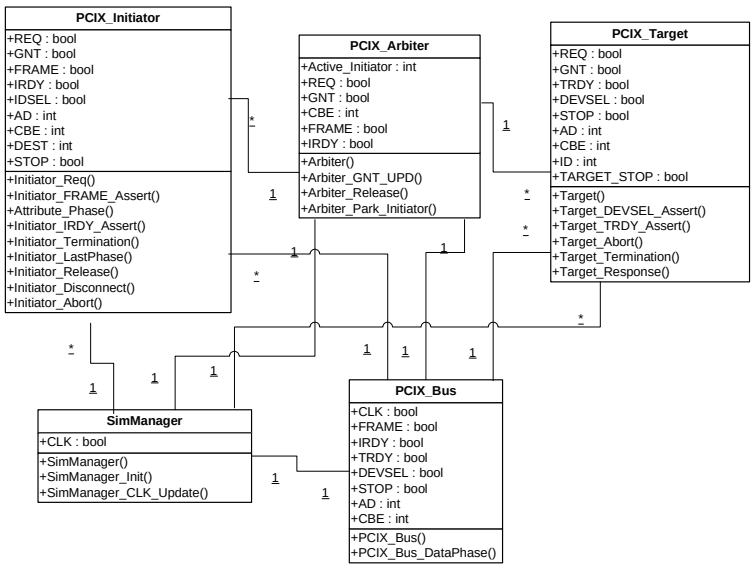

Fig. 6. Class Diagram of PCI-X.

is one-to-many (1 - *) because there is only one arbiter which is connected to many initiators. However, the relationship between the SimManager and arbiter is one-to-one (1 - 1) as there is only one SimManager and one arbiter in the system.

We modeled different modes and types of operations of PCI-X using sequence diagrams. Sequence diagrams enable us to model the bus in AsmL easily and efficiently. Also, they help to extract the properties of the system being modeled, which can be used to verify using model checking. In addition, UML modeling helps to close the gap between informal specification and formal models in AsmL.

In Figure 7, we show the protocol sequence of a typical Mode 1 transaction of the PCI-X using a sequence diagram. Figure 7 is a best case scenario of Mode 1 transactions, with one initiator and one target and without any wait states. In the first clock cycle, the initiator asserts the REQ# signal to get the control of the bus by executing the methods *Initiator_Req()*. The arbiter asserts the GNT# signal to that initiator through *Arbiter_GNT_UPD()*. In the third clock cycle, the address phase takes place and also the FRAME# signal is asserted by the initiator to signal the start of the transaction using the method *Initiator_FRAME_Assert()*. In the next clock cycle, the attribute phase takes place, where additional information included with each transaction is added. In clock cycle N+4, the DEVSEL# signal is asserted by the target using *Target_DEVSEL_Assert()* and in the next clock cycle, the data phase is started with the assertion of the IRDY# (*Initiator_IRDY_Assert()*) and TRDY# (*Target_TRDY_Assert()*) signals by the initiator and the target, respectively. Before the last data phase, the FRAME# signal is deasserted using *Initiator_Termination()* to signal the completion of the data transfer

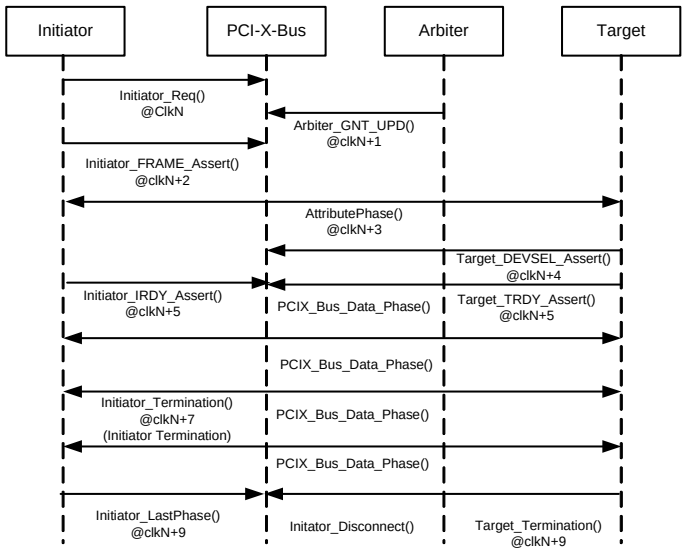

Fig. 7. Sequence Diagram of Mode 1 Transaction.

and in the termination phase all the other signals are de-asserted. In order to represent the clock constraints of the PCI-X transaction, we added an additional operator, "@", to specify at which clock cycle a particular action should occur.

Figure 8 shows the protocol sequence of a typical Mode 2 transaction of PCI-X using a sequence diagram. Mode 2 transaction is pretty similar to Figure 7, except that there is an additional delay of one clock cycle (*Target Response Phase*) and one additional idle clock between any two transactions.

In Figure 9, we show the transaction sequences of a 16 bit interface of Mode 2 transaction. In this transaction, the attribute phase takes two cycles unlike the transaction types. Other signal activities of this transaction are the same as the generic Mode 2 transaction. In the coming section, we present the AsmL modeling of PCI-X from UML.

5.3. *AsmL Modeling of PCI-X Bus*

We use AsmL's class features to model all the core components of PCI-X. Each of these has its own data members (signals) and methods (behavior) in addition to the constructors. We also use enumeration features (*enum*) of AsmL to model different modes of PCI-X, different types of transaction phases, the state of the system and the clock.

In addition, we exhaustively use the *require* and "*:=*" statements of AsmL in our design approach. *require* is the pre-condition statement in AsmL used to check if a certain condition is satisfied in order to enable a method, and the operator "*:=*" represents an update statement used to change the system state. Figure 10 shows

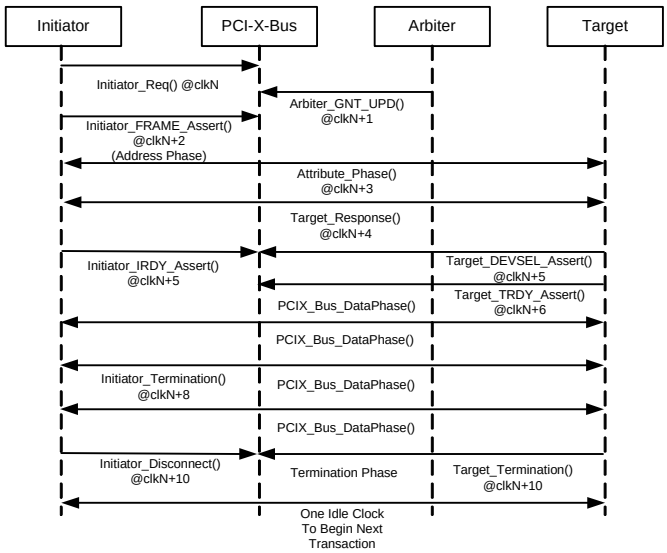

Fig. 8. Sequence Diagram of Mode 2 Transaction.

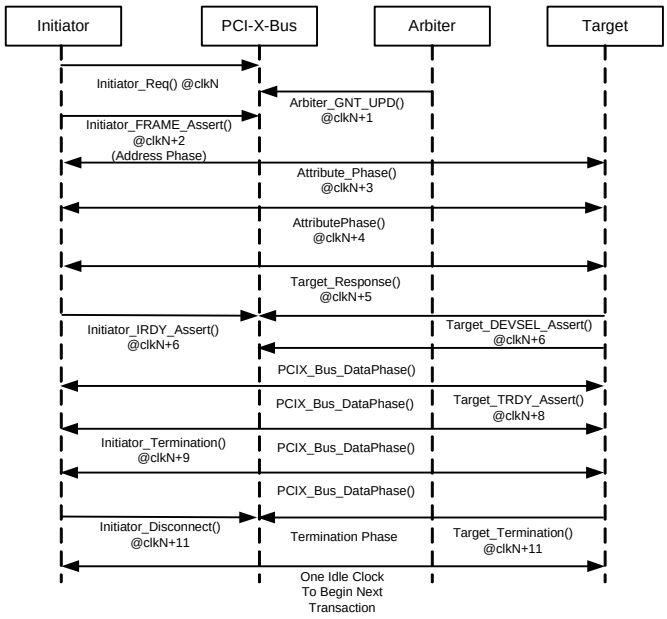

Fig. 9. Sequence Diagram of Mode 2 Transaction (16 bit).

the AsmL model of the arbiter grant method (*Arbiter_GNT()*). As the name of the method says it acts as an arbiter for granting the bus to the requesting initiator. In

order to grant the bus to the requested initiator, this method has the following pre-conditions (*require*) to be met: (1) there must be at least one initiator requesting the bus and that initiator has not been granted the bus at the time of the request; (2) the clock is on the rising edge; and (3) the mode can be either Mode 1 or Mode 2. If these pre-conditions are met, the arbiter updates the GNT# signal.

```
public Arbiter_GNT()
  require (exists x in Initiators where x.REQ =  true and
          x.GNT = false) and  me.GNT = false and Smanager.CLK = CLK_UP
            and (Mode = MODE_1 or Mode = MODE_2)
  me.Active_Initiator := min y | y in Initiators_Range where
           (Initiators(y).REQ = true)
  me.GNT := true
  Initiators(Active_Initiator).GNT := true
```

Fig. 10. Arbiter Grant AsmL Method.

In Figure 11, we show how a target can signal its readiness using TRDY# signal. We call this method as *Target_TRDY_Assert()*. The pre-conditions are the following: TRDY# is *false*, FRAME# and DEVSEL# are *true*, CLK is *CLK_UP*, Phase is *DATA_PHASE_FIRST* and the *AD* of the Bus should be the *ID* of the target. If the pre-conditions are true, then TRDY# will be asserted.

```
public Target_TRDY_Assert()
    require me.TRDY = false and Bus.FRAME = true and
            Bus.AD = me.ID and Bus.DEVSEL = true and
            Smanager.CLK = CLK_UP and Phase = DATA_PHASE_FIRST
    me.TRDY := true
    me.AD := Bus.AD
    Bus.TRDY := true
    Phase := DATA_PHASE
```

Fig. 11. Target Assert AsmL Method.

Figure 12 shows the initiator termination method (*Initiator_Termination*). This method basically signals the end of a transaction if *BYTECOUNT* is less than 2. *BYTECOUNT* indicates the number of bytes of data transfer in a transaction. This signaling is done by de-asserting the FRAME# signal and updating the transaction phase to the initiator termination phase (INI_TER_PHASE). This method has specific pre-conditions that need to be true so that it can terminate the transactions. The pre-conditions (*require*) are the following: initiator's REQ#, GNT#,

FRAME#, IRDY#, DEVSEL#, TRDY# are asserted, *BYTECOUNT* is less than 2, and the clock is on the rising edge. If all above pre-conditions are true, this method updates the FRAME# signal to false and the phase to *INI_TER_PHASE*. After this FRAME# signal de-assertion, the initiator's last phase method (*Initiator_LastPhase()*) is invoked.

```
public Initiator_Termination()
  require me.GNT = true and me.REQ = true and me.FRAME = true and
          me.IRDY = true and Bus.TRDY = true and Bus.DEVSEL = true
          and BYTECOUNT < 2 and me.STOP = false and
          Smanager.CLK = CLK_UP
  me.FRAME := false
  Bus.FRAME := false
  Phase := INR_TER_PHASE
```

Fig. 12. Initiator Termination AsmL Method.

5.4. *PCI-X Bus PSL Properties*

Properties are embedded in every state in the FSM generated by the Asmlt and is represented by two Boolean state variables P_{eval} and P_{val}. P_{eval} represents whether a property can be evaluated or not and P_{val} denotes the state of the property in the current state. A violated property is detected $P_{val} = false$.

We define various properties of the PCI-X bus in PSL. The properties are obtained from the sequence diagrams and the informal specification. Some of the properties that we show are generic to any bus protocols and others are specific to PCI-X. The properties range between common behaviors for any bus standard to specific properties for the PCI-X high speed bus standard. We defined 10 PSL properties and then modeled them in AsmL. The description and AsmL code for these properties is shown in Appendix A.

5.5. *Experimental Results*

Table 2 details the results of model checking the aforementioned 10 properties (see Appendix A) on a PCI-X model with 5 initiators and 5 targets. The informal descriptions and formal definition of these properties in AsmL is given in Appendix A. In the table, we show the CPU time, number of states and transitions for the PCI-X model with the various properties defined. The experiments were performed on a Pentium IV processor (2.4 GHz) with 768 MB of memory. Even though, the CPU time to verify the properties is relatively short, we also have to consider the time spent to write the properties and to learn the tool. Using the approach proposed in [7], it is also possible to model check a SystemC design of the PCI-X, but

Table 1. Model Checking Results

Property	CPU Time (s)	States	Transitions
P1	385.24	2169	3250
P2	194.23	1800	2563
P3	150.52	1578	2156
P4	130.45	1489	2096
P5	156.35	1478	2265
P6	173.50	1925	2439
P7	174.47	2013	2698
P8	178.42	1873	2359
P9	256.63	2192	2980
P10	143.52	1356	1923

the state space explosion will be worse considering the complexity of the SystemC simulation semantics and the OO nature of the language. The results achieved here are superior to the results obtained in [7] in terms of machine execution time, and taking in consideration that the time here represents both the generation of the FSM and the evaluation of the property, which combines the time for the two separated steps in the other approach: generating the FSM and model checking using an external model checker. We also presented a high level model for the PCI-X bus and verified several nontrivial properties illustrating the efficiency of our approach.

Comparing the results we obtain in our approach to previous works in [3,21,24,27,29], this work considers designing and verifying the latest high-speed bus standard (PCI-X) including its very complex transaction rules. In addition, the modeling of the PCI-X at the transaction level and verifying various properties using model checking approach is made feasible using our methodology.

6. Conclusion

The verification of systems is the bottleneck in the design cycle because systems combine various hybrid components and behaviors. Classical functional verification is consuming an inordinate amount of the design cycle time. This paper presents an integration of a design for verification approach into the high level design process and enables model checking for large systems based on embedding of PSL in AsmL. Starting from an informal specification of the system, we derive first an UML description in terms of class and sequence diagrams to validate high level behavior. From the UML design, we derive an AsmL model of the system refining more details

of its implementation, and which we formally verify against a set of PSL properties. The verification of PSL properties is performed by including the PSL properties in the AsmL design and generating an FSM of the model that contains the embedded PSL properties. Using Asmlt, we perform state exploration which gives the notion of model checking. For embedding of the Property Specification Language (PSL) in Abstract State Machines Language (AsmL), we adopted 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.

We illustrated this approach through a case study of the PCI-X bus, on which we verified several PSL sample properties. The PCI-X bus is an industrial size design. The UML representation of PCI-X was developed in terms of class diagrams and sequence diagrams. Then, an AsmL model was designed from the UML representation. We then integrated various PSL properties into the AsmL model and applied model checking for the system. We achieved promising results by verifying several properties on the bus model. As future work, we intend to provide a mechanized refinement approach by generating SystemC model from the AsmL model. It is also possible to formalize 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 [25].

References

1. E. Börger and R. Stärk. Abstract State Machines. A Method for High-Level System Design and Analysis. Springer–Verlag, 2003.
2. K. Claessen and J. Martensson. An Operational Semantics for Weak PSL. In Formal Methods in Computer-Aided Design, LNCS 3312, Springer–Verlag, pp. 337–351. November 2004.
3. K. H. Chang, Y. C. Su, W. T. Tu, Y. J. Yeh, and S. Y. Kuo. A PCI-X Verification Environment Using C and Verilog. In VLSI Design/CAD Symposium, Taiwan, 2003.
4. M. Chong. A PCI Express to PCI-X Bridge Optimized for Perfromance and Area. Master's thesis, Department of Electicial Engineering and Computer Science, Massachussets Institute of Technology, March 2004.
5. M. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. Pasareanu, and R. W. Visser amd H. Zheng. Tool-supported Program Abstraction for Finite-state Verification. In International Conference on Software Engineering, pp. 177-187, Toronto, Canada, 2001.
6. A. Gawanmeh, A. Habibi and S. Tahar. Enabling SystemC Verification using Abstract State Machines. In Languages for Formal Specification and Verification, Forum on Specification & Design Languages, September 2004.
7. A. Gawanmeh, A. Habibi, and S. Tahar. Embedding and Verification of PSL using ASM. In IEEE International Workshop on System-on-Chip, pp. 125–130, December 2006.
8. A. Gawanmeh, S. Tahar and K. Winter. Formal verification of ASMs using MDGs. To appear in Journal of Systems Architecture, Elsevier Science Publishers, 2008.
9. 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.

10. D. Grobe and R. Drechsler. Checkers for SystemC Designs. In Second ACM & IEEE International Conference on Formal Methods and Models for Codesign, pages 171178, San Diego, USA, 2004.
11. M. Gordon, J. Hurd and K. Slind. Executing the Formal Semantics of the Accellera Property Specification Language by Mechanised Theorem Proving. In Correct Hardware Design and Verification Methods, LNCS 2860, Springer–Verlag, pp. 200–215, October 2003.
12. M. Gordon and T. Melham. Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic. Cambridge, U.K., Cambridge Univ. Press, 1993.
13. M. Gordon. Validating the PSL/Sugar Semantics Using Automated Reasoning. Formal Aspects of Computing, 15(4): 406–421, 2003.
14. Y. Gurevich. Evolving Algebras 1995: Lipari Guide. In Specification and Validation Methods, Oxford University Press, 1995.
15. Y. Gurevich, B. Rossman, and W. Schulte. Semantic Essence of AsmL. Technical report, Microsoft Research, MSR-TR-2004-27, March 2004.
16. Ali Habibi. A Framework for System Level Verification: The SystemC Case. Ph.D. Thesis. Concordia University, Montreal, Canada, November 2005.
17. A. Habibi and S. Tahar. On the Transformation of SystemC to AsmL using Abstract Interpretation. In Electronic Notes in Theoretical Computer Science, Volume 131: 39–49, 2005.
18. J. Huggins. Abstract State Machines website. http://www.eecs.umich.edu/gasm, 2003.
19. M.L. McMillan. Symbolic Model Checking, Kluwer Academic Pub., 1993.
20. H. Moinudeen, A. Habibi, and S. Tahar. Design for Verification of the PCI-X Bus. In IEEE International Conference on Formal Methods in Computer-Aided Design, IEEE Computer Society Press, pp. 187-188, November 2006.
21. K. Oumalou, A. Habibi, and S. Tahar. Design for Verification of a PCI Bus in SystemC. In Symposium on System-on-Chip, IEEE Computer Society Press, pp. 201–204, Tampere, Finland, November 2004.
22. PCI Special Interest Group. www.pcisig.com, 2005.
23. Accellera Property Specification Language Reference Manual, Version 1.01. http://www.accellera.org, 2004.
24. K. Shimizu, D. Dill, and A. Hu. Monitor-based formal specification of PCI. In Formal Methods in Computer-Aided Design, LNCS 1954, Springer-Verlag, pp. 335–353, 2000.
25. SystemVerilog. http://www.systemverilog.org, 2004.
26. C. Wallace. The Semantics of the C++ Programming Language. In Specification and Validation Methods, Oxford University Press, 1995, pages: 131–164.
27. R. Wang and Z. Wen. A Verification Environment for PCI-X BFMs in VERA. Technical report, Synopsys Inc., 2002.
28. K. Winter. Model Checking Abstract State Machines, Ph.D. thesis, Technical University of Berlin, Germany, 2001.
29. C. C. Yu, K. Chang, Y. Yeh, and S. Kuo. System Level Assertion-Based Verification Environment for PCI/PCI-X and PCI-express. In VLSI Design/CAD Symposium, Taiwan, 2004.

Appendix A. PSL Properties for PCI-X Bus in AsmL

The first property *P1*, is a common behavior for any bus standard. It states that if an initiator is requesting the bus *(!Initiator.REQ == true)*, it will eventually be granted *(!Initiator.GNT == true)*. This also makes sure the fact that no initiator will be using the bus indefinitely. It is to be noted that all signals of PCI-X are active-low.

```
Property P1:
  forall Initiator in {Initiator0, ..., Initiator4}
    if(!Initiator.REQ == true) then
      eventually (!Initiator.GNT == true)
```

Property *P2* is about the termination of a PCI-X transaction be it Mode 1 or Mode 2. It means that if an initiator is terminating the bus by asserting the *STOP* signal, then eventually the bus will be released by de-asserting the *FRAME* signal and targets will be released by de-asserting *TRDY* and *DEVSEL*.

```
Property P2:
  forall Initiator in {Initiator0, ..., Initiator4}
    if((!Initiator.STOP == true) and
       (!Initiator.GNT == true)) then
      eventually {(!Bus.FRAME == false) and
        forall Target in {Target0, ..., Target4}
          (!Target.TRDY == false) and
          (!Target.DEVSEL == false)}
```

Property *P3* is for the assertion of *FRAME* signals. This property is important for the start of the transaction. If an initiator is granted to the bus, then in the next clock cycle the *FRAME* signal has to be asserted.

```
Property P3:
  forall Initiator in {Initiator0, ..., Initiator4}
    if(!Initiator.GNT == true) then
      next[0] (!Initiator.FRAME == true)
```

Property *P4* is to check the phase *(ADDR_PHASE, ATTR_PHASE, IDLE_PHASE)* of the transaction. If the initiator FRAME signal is asserted, then in the next clock cycle, the phase of the transaction will be the attribute phase (ATTR_PHASE).

```
Property P4:
  forall Initiator in {Initiator0, ..., Initiator4}
    if(!Initiator.FRAME == true) then
      next[0] (Transaction_Phase == ATTR_PHASE)
```

Property *P5* is regarding the arbitration of the bus. If an initiator is selected by the arbiter, then it will be able to get access to the bus by setting *!Bus.FRAME*. Then, its destination target will be activated by setting its *!Target.TRDY* and the initiator will release the bus once *!Initiator.GNT* is set to false.

```
Property P5:
  forall Initiator in {Initiator0, ..., Initiator4}
  forall Target in {Target0, ..., Target4}
   if((!Target.GNT == true) and
       (!Initiator.DEST == Target.ID)) then
    eventually {(!Bus.FRAME == true) and
        (!Initiator.TRDY == true) and
        (!Target[ID].TRDY == true) and
        (!Target.GNT == false)}
```

Property *P6* is to check the response of a target when it has been selected as destination. This property is specific for Mode 1 transaction. The property says that if the transaction mode is MODE_1 and the initiator's FRAME is asserted, then eventually the *IRDY* and *TRDY* of the initiator and target respectively, will be asserted.

```
Property P6:
  forall Initiator in {Initiator0, ..., Initiator4}
  forall Target in {Target0, ..., Target4}
    if((!Initiator.FRAME == true) and
      (Transaction_Mode == MODE_1)) then
      eventually {(!Initiator.IRDY == true) and
        (!Target.TRDY == true)}
```

Property *P7* is related to the Mode 2 transaction. If *FRAME* is asserted and transaction is Mode 2 then eventually *IRDY* and *DEVSEL* will be asserted together.

```
Property P7:
  forall Initiator in {Initiator0, ..., Initiator4}
  forall Target in {Target0, ..., Target4}
    if(!Initiator.FRAME == true) and
      (Transaction_Mode == MODE_2) then
      eventually {(!Initiator.IRDY == true) and
        (!Target.TRDY == true)}
```

Property *P8* is about the target response property of Mode 2 transaction. If *FRAME, IRDY, DEVSEL* are asserted and transaction is Mode 2, then in the next clock cycle, *TRDY* will be asserted.

```
Property P8:
  forall Initiator in {Initiator0, ..., Initiator4}
  forall Target in {Target0, ..., Target4}
    if(!Initiator.FRAME == true) and
      (!Initiator.IRDY == true) and
      (!Target.DEVSEL == true) and
      (Transaction_Mode == MODE_2) then
       next (!Target.TRDY == true)
```

Property *P9* checks the idle phase of Mode 2 transaction after a transaction has completed. It says that if *IRDY, STOP, DEVSEL, TRDY* are de-asserted then in the next clock cycle, the phase of the transaction will be idle. In other words, the bus will be idle.

```
Property P9:
  forall Initiator in {Initiator0, ..., Initiator4}
  forall Target in {Target0, ..., Target4}
    if((!Initiator.IRDY == false) and
      (!Initiator.STOP == false) and
      (!Target.DEVSEL == false) and
      (!Target.TRDY == false) and
      (Transaction_Mode == MODE_2)) then
       next (Transaction_Phase == IDLE_PHASE)
```

Property *P10* is about the initiation of Split transaction. In split transaction, the target can request for the bus *(!Target.REQ == true)* and it can be eventually granted *(!Target.GNT == true)*.

```
Property P10:
  forall Target in {Target0, ..., Target4}
    if(!Target.REQ == true) and
      (Transaction_Mode == SPLIT) then
      eventually (!Target.GNT == true)
```