

Design and Verification of SystemC Transaction Level Models

Ali Habibi and Sofiène Tahar

Department of Electrical and Computer Engineering,

Concordia University, Montreal, Canada

Email: {habibi, tahar}@ece.concordia.ca

Technical Report

December 2004

Abstract

Transaction level modeling allows exploring several SoC design architectures leading to better performance and easier verification of the final product. In this report, we present an approach to design and verify SystemC models at the transaction level. We integrate the verification as part of the design-flow where we first model both the design and the properties (written in PSL) in UML; then, we translate them into an intermediate format modeled with AsmL (language based on Abstract State Machines (ASM)). The AsmL model is used to generate an FSM of the design including the properties. Finally, we translate the verified design to SystemC and map the properties to a set of assertions (as monitors in C#) that can be reused to validate the design at lower levels by simulation. At the SystemC level, we also present a genetic algorithm to enhance the assertions coverage. We will ensure the soundness of our approach by proving the correctness of the SystemC to AsmL and AsmL to SystemC transformations. We illustrate our approach on two case studies including the PCI bus standard and a Master/Slave generic architecture from the SystemC library.

Contents

1	Introduction	4
2	Design Methodology	6
2.1	Modeling PSL Properties	7
2.1.1	UML Model	7
2.1.2	AsmL Model	8
2.2	Modeling SystemC	10
2.2.1	AsmL Model	10
2.2.2	Translation to SystemC	11
3	Verification Methodology	12
3.1	Model Checking	12
3.2	Assertion Based Verification	13
3.3	Assertions' Coverage Enhancement	15
3.3.1	Static Code Analysis	15
3.3.2	Genetic Algorithm	16
4	Correctness of the SystemC/AsmL and AsmL/SystemC Transformations	17
4.1	SystemC Fixpoint Semantics	18
4.1.1	Syntactical Domains	18
4.1.2	Fixpoint Semantics	19
4.2	AsmL Fixpoint Semantics	21
4.2.1	Syntactical Domains	21
4.2.2	Fixpoint Semantics	22
4.3	Program Transformation	23
5	Experimental Results	25
5.1	Models Description	25
5.2	Model Checking	25
5.3	Assertion Based Verification	26
5.4	Assertions Coverage Enhancement	27
6	Related Work	29
7	Conclusion	30

1 Introduction

SystemC [25] 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, DSPs, etc.) communicate among each other over shared resources.

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 report, 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 at a high level of abstraction allowing easier validation and verification operations. We wrote the ASM models in the AsmL language [21] one of the very latest languages developed for ASM [9] supporting object-oriented modeling at higher level of abstraction in comparison to C++ or Java. Besides, using the AsmL Tester, from an AsmL specification it is possible to generate finite state machines (FSMs).

To enable the integration of both the model and the properties at the ASM level, we embedded the PSL semantics in AsmL. 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 (using the AsmL compiler [21]) to C# while we translate the design from AsmL is translated SystemC. Both codes, SystemC and C#, are then combined to form a single model enabling the verification of the assertions by simulation.

The objective of the verification process is not only to write assertions but to verify them. This latter task is usually performed using test vectors generation tools mostly based on random processes. This kind of blind simulation does not guarantee that the assertion will be covered during the test execution. Therefore, it is very important to consider a smarter and more efficient test vector generation approach. To do so, we propose first to use static code analysis to extract a dependency relation between the design inputs and the assertion's variables. This analysis will also define for every input the range of possible values that may trigger the assertion which provides some very useful information to improve the assertion's coverage.

In order to enhance the coverage even more, we also propose to use a genetic algorithm based on a community of random generators having a variety of DNA information [5]. This latter will help defining the list of variables considered in the test generation, their possible values and a weighted probability over the previous range [13]. The DNA update/mutation rules will be defined according to the coverage each generator offers. At the end of the genetic procedure, we expect the final DNA to provide an identification of a generator that offers a better coverage than a random one.

The soundness of our approach relies on proving the correctness of the SystemC to AsmL and vice-versa transformations. The basic concept of this proof of soundness is based on the systematic design of program transformation frameworks defined in [3]. For instance, we provide a formalization of the SystemC and AsmL semantics in fixpoint based on the OO general case given in [20]. Then, we prove that, for every SystemC (respectively AsmL) program, there exists an AsmL (respectively SystemC) program preserving the same properties, w.r.t. an observation function α_o .

The rest of this report is organized as follows: Section 2 describes the proposed design methodology. Section 3 discusses the proposed verification approach. Section 4 presents the proofs of correctness of our approach. Section 5 illustrates our approach on two case studies. Section VI discusses the related work. Finally, Section 7 concludes the report.

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 an ASM model in AsmL 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. Both tasks, UML update and UML to AsmL translation are repeated until all the properties pass (either proved to be correct 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 AsmL is, from the other side, translated to SystemC and co-integrated with the assertions for verification by simulation.

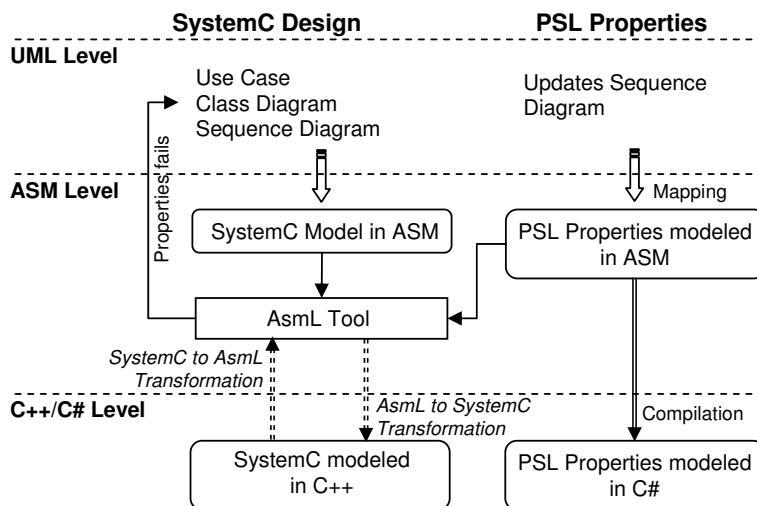


Figure 1: Design and Verification Methodology.

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 [4] or as top layer in combination with existent languages (such as SystemC)[28]. Nevertheless, these proposals neglected totally considering the properties of the system (PSL like properties in particular) while sequence diagrams, for example, include very useful information to set transaction properties for TLM in particular [12].

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 PSL. In fact, PSL was defined for a real instances formed from objects, from the design, 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.

Number of cycles: every action can include the information about after how many cycles the method will 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.

Method duration: certain methods are supposed to execute for a certain number of cycles (for e.g., reading cycle takes four cycles). We added an operator \$ to specify this 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 fourth cycle and a notification will be eventually sent by the slave to the bus who will forward it in the next cycle to the Master.

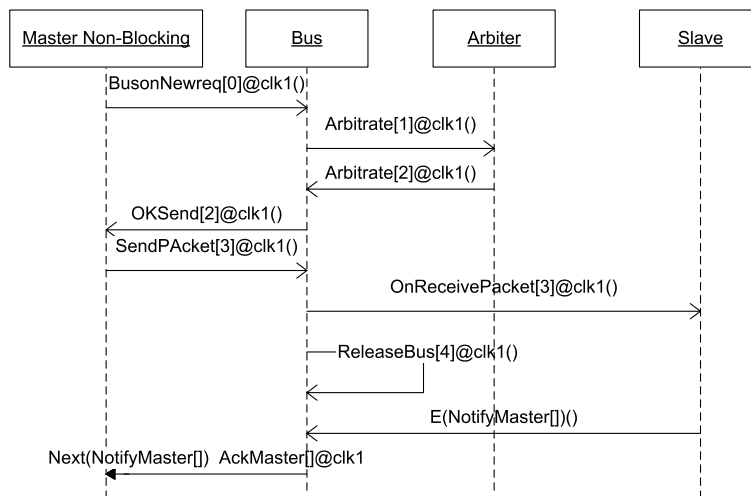


Figure 2: Example of a Modified UML Sequence diagram.

2.1.2 AsmL Model

There are two ways to embed PSL properties into the design, either as part of the design code itself or by adding them as external monitors. We adopted the second 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, which can be 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 [12].

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 [1], and the formal semantics definition in HOL [6]. Our 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.
- (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 two methods: a method providing the previous values of a variable (e.g., *prev()*) and a method providing 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:

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, etc.). 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 a set operations to add,remove and update the unit's list of properties.


```

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 AsmL.

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 AsmL Model

Our design model at the ASM level is purely OO 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 part of AsmL tool. So, a specific style of programming is required in addition to a precise configuration of the algorithm. This latter 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.

For a model M including a set of classes, $C = \{c_1, \dots, c_n\}$, 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^{meth} 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_{FSM}^1 : 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_{FSM}^2 : 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_{FSM}^3 : 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 ensures that in every state we only explore the involved methods.

Rule R_{FSM}^4 : 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 AsmL 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 SystemC

Once the AsmL 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 AsmL 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_{C++}^1 : "Basic types": AsmL 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 AsmL Model.

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

Rule $R_{C++}^{2.1}$: "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 $R_{C++}^{2.2}$: "Class Methods": in AsmL 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 AsmL 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_{C++}^3 : "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 SystemC (C++)/C# level.

3.1 Model Checking

PSL properties are embedded in AsmL 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, \dots, x_n , and y_1, y_2, \dots, y_m belonging to the Boolean layer, and the sequences, S_1 and S_2 belonging to the temporal layer, we can define: $S_1 = \{x_1, x_2, \dots, x_n\}$, and $S_2 = \{y_1, y_2, \dots, y_m\}$ and then use assertions to check any PSL operation between S_1 and S_2 such as $S_1 \text{ OP } S_2$, where OP is a PSL operator (e.g., implication ($:$), or equivalence (\Leftrightarrow)). The assertion is built as follows:

1. Add all the Boolean items to the sequences:

$$\forall i \text{ in } 1 \text{ to } n : S_1.\text{AddElement}(x_i) \quad \forall j \text{ in } 1 \text{ to } m : S_2.\text{AddElement}(y_j)$$

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

3. Define the *verification unit* as an assertion, A , that includes the property P :
 $A.\text{Add}(P)$

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

Figure 5 describes our methodology to integrate and verify PSL assertions for SystemC designs, which consists of the following three main steps: (1) Updating the SystemC design to interface to the assertion monitor; (2) Generating the assertion as a C# code from its AsmL description; and (3) Integrating the assertion in the design.

Generating the table of symbols from the SystemC design is important in order to validate the variables (names and types) that are used in the assertion. While compiling the assertion, we are concerned with, first, its syntactical correctness, and second, its semantical validity where we check the type and the naming of the assertion variables.

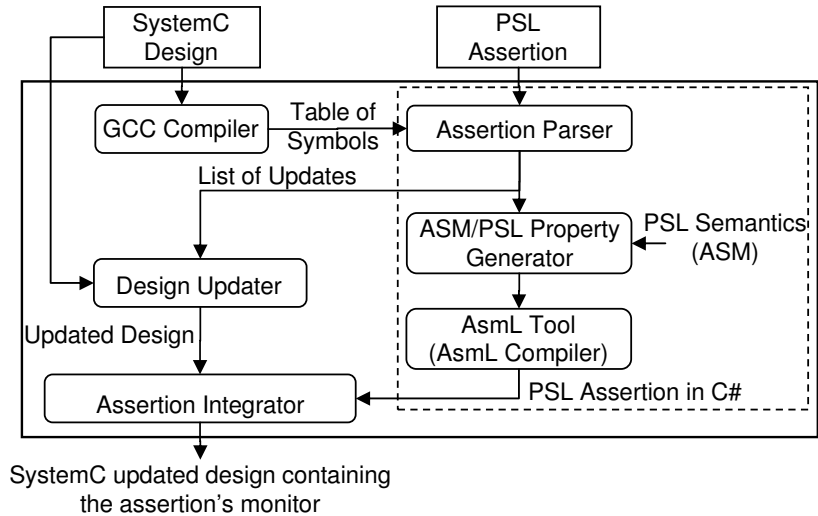


Figure 5: Methodology to verify PSL assertions for SystemC designs

Once the assertion’s structure verified, we translate it to its equivalent AsmL code. In our embedding of the assertion in AsmL, we defined a one to one mapping between the PSL assertion and their AsmL embedding. Hence, the transformation is purely syntactical, which guarantees the correctness of the embedded assertion.

In the validation phase of the assertion structure, we also generate a list of updates required to prepare the design to integrate the assertion. For instance, the signals (variables) that are used in the assertion have to be input to the assertion monitor. For this reason, we provide the Design Updater with a list of variables as defined by their unique identifier in the table of symbols. Then, the Design Updater modifies the SystemC design to make the needed variables visible to the monitor. This transformation does not affect the behavior of the code because these variables will be accessed in a read-only mode.

Once the code is updated and the assertion is generated, the Design Integrator will 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#.

3.3 Assertions' Coverage Enhancement

Once the assertion is integrated with the design, our next goal is to define a test generation approach that offers better coverage of the assertions. To do so, we first start by statically analyzing the design in order to define a dependency relation between the system inputs and the assertions variables. Such a relation is very useful to omit the inputs that are not affecting the assertion. It serves also identifying the required inputs and the range of their possible values that may affect the assertion. We also identify which processes need to be activated in order to get the assertion fired. Figure 6 gives an overview of our methodology, including the following steps:

1. *Static Analysis*: We apply a static analysis technique to generate an abstract representation of the design modeled as graph, called *hypergraph* [29], that will include a representation of both the program's environment and the process's environment.
2. *Dependency check*: From the hypergraph representation, we extract the dependency graph and the range of inputs that may affect the assertion.
3. *Test Program generator*: Using the abstract program (modeled as a hypergraph structure) and the dependency graph, we generate a reduced model containing only the units involved in the assertion.
4. *Initial DNA generation*: Considering the list of input variables of interest for the assertion and their ranges, we create a DNA structure that will serve as starting point for the genetic algorithm.
5. *DNA evaluation/update*: Using the initial DNA, the algorithm will update the generators' community starting from the initial DNA to obtain an optimal DNA using the assertion coverage as selection criteria.

3.3.1 Static Code Analysis

In order to analyze SystemC designs statically and extract the required information to generate the "inputs/assertions variables" dependency relation, we considered an approach based on abstract interpretation [2], a formal technique that has proven to be efficient with object-oriented languages and large programs.

At the end of the analysis, the program is represented as a hypergraph [29], which can be interpreted as a general automata connecting its states by branches (also called hyper-branches). These branches can be seen as an extension to Binary Decision Diagrams (BDDs), but more adapted to programs representation. We augmented this work to support the SystemC library and simulator in the form

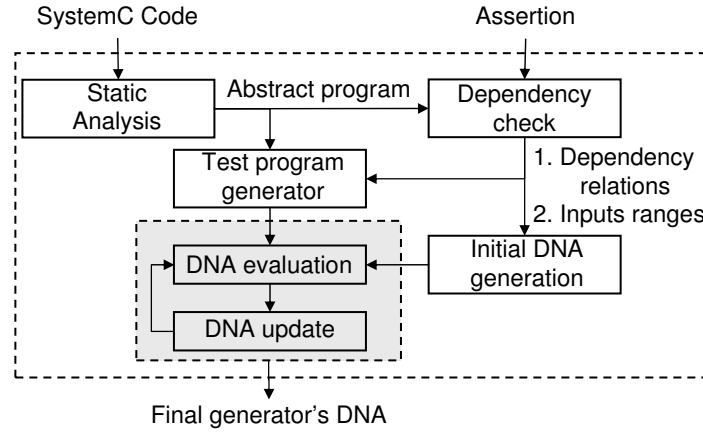


Figure 6: Enhancing the Assertion’s Coverage.

of specific classes to extract information related to SystemC processes and events from the design [17].

3.3.2 Genetic Algorithm

Genetic algorithms belong to a family of computational models inspired by evolution [18]. They encode a potential solution to a specific problem on a simple chromosomes like data structure and apply recombination operators to these structures to preserve critical information. Since their introduction by Holland [18], genetic algorithms have been applied to a broad range of learning and optimization problems [26]. Typically, a genetic algorithm starts with a random population of encoded candidate solutions (test generators for our case), called chromosomes. The objective is to maximize the likelihood of generating an optimal solution. This can be guaranteed by: (1) evaluating the *fitness* of each candidate solution in the current population; (2) selecting the fittest candidate solutions to act as parents of the next generation of candidate solutions; and (3) selected parents are recombined and mutated to generate offsprings.

In our context, the search space to be explored is the state space of the system that may trigger the assertion(s) under verification. Candidate solutions are finite sequences of input ranges and probability weights. Each candidate solution is encoded by a chromosome (a finite string of bits). The information encoded in the DNA includes: (1) the list of input variables, (2) their ranges (possible values), and (3) a weighted probability to their random generation. The algorithm evaluates the fitness of the candidate by executing a test generation based on the information

embedded in the corresponding chromosome. A coverage report is then generated to serve in the fitness evaluation phase.

The chromosome encoding is the most important aspect of our algorithm. During the static analysis phase, we obtain the list of variables of the program and their types. Each variable is given a unique identifier. Each type is also given a space of possible values (for the type *char* for example the range is [0..255]). The chromosome encodes the list of variables, their types and a weight relation over the range of possible values. This latter varies according to the type and its interpretation. For every basic type, we defined a list of possible weight relations, e.g., for *Integer*, we use the following window relation: $I < -50$ or $I > 50$ for $w = 0.2$ and $-50 \leq I \leq 50$ for $w = 0.8$.

This relation states that the integer variable I is generated randomly in the interval [-50, 50] with a probability of 80% and 20% inside and outside the interval, respectively.

The proposed fitness function serves to guide the genetic search towards firing the assertion's variables. Its intuitive idea is to reduce the range of possible values of the input variables and to find the best probability distribution of the random test generation that will modify the assertion's variables. This way, we maximize the assertion evaluations, since the evaluation of the chromosomes is defined as an award bonus proportional to the number of assertion evaluations. In order to improve the efficiency of the algorithm, we keep track of the best and worst chromosome fitness in each generation; if both fitness values become equal, we increase the mutation rate, in order to help the genetic evolution get out of local maxima. Once there is an improvement in the overall fitness, we restore the original mutation rate to continue the evolution normally.

4 Correctness of the SystemC/AsmL and AsmL/SystemC Transformations

The work of Patrick and Radhia Cousot in [3] is the essence for any program transformation using abstract interpretation. The tactical choice of using semantics to link the subject program to the transformed program is very smart in the sense that it enables proving the soundness proof of the transformation, related to an observational semantics. The transformation from SystemC to AsmL, and vice-versa, represents an online program transformation which corresponds to the approach described in Section 3.9 of [3]. Figure 7 displays a projection of that generic methodology on a SystemC subject program and an AsmL transformed

program. The same figure can be used to perform the soundness of a transformation and also to construct it. In both cases, we need to define the syntax, semantics and observation functions for both AsmL and SystemC.

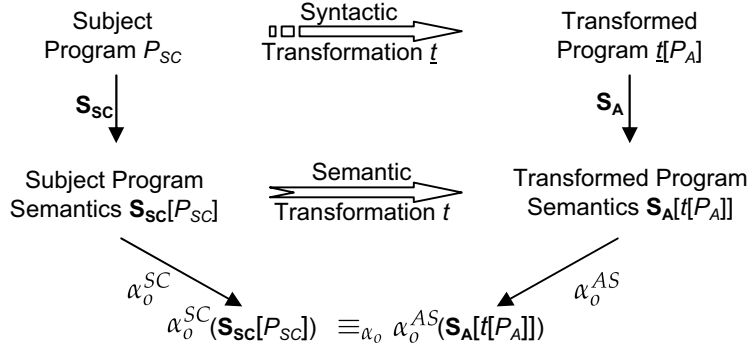


Figure 7: Online Program Transformation.

4.1 SystemC Fixpoint Semantics

4.1.1 Syntactical Domains

SystemC have a large number of syntactical domains. However, they are all based on the single `SC_Module` domain. Hence, the minimum representation for a general SystemC program is as a set of modules.

Definition 4.1 (*SystemC Module: SC_Module*)

A SystemC Module is a set $\langle DMem, Ports, Chan, Mth, SC_Ctr \rangle$, where *DMem* is a set of the module data members, *Ports* is a set of ports, *Chan* a set of SystemC *Chan*, *Mth* is a set of methods (functions) definition and *SC_Ctr* the module constructor.

Definition 4.2 (*SystemC Port: SC_Port*)

A SystemC Port is a set $\langle IF, N, SC_In, SC_Out, SC_InOut \rangle$, where *IF* is a set of the virtual methods declarations, *N* is the number of interfaces that may be connected to the port, *SC_In* is an input port (provides only a *Read* method), *SC_Out* is an output port (provides only a *Write* method) and *SC_InOut* is an input/output port (provides *Read* and *Write* methods).

In contrast to default class constructors for OO languages, the SystemC module constructor `SC_Ctr` contains the information about the processes and threads that will be executed during simulation.

Definition 4.3 (*SystemC Constructor: `SC_Ctr`*)

A SystemC Constructor is a set $\langle \text{Name}, \text{Init}, \text{SC_Pr}, \text{SC_SSt} \rangle$, where *Name* is a string specifying the module name, *Init* is a default class constructor, *SC_Pr* a set of processes and *SC_SSt* is a set of sensitivity statements (to set the process sensitivity list *SC_SL*).

Definition 4.4 (*SystemC Process: `SC_Pr`*)

A SystemC process is a set $\langle \text{PMth}, \text{PTh}, \text{PCTh} \rangle$, where *PMth* is a method process (defined as a set $\langle \text{Mth}, \text{SC_SL} \rangle$ including the method and its sensitivity list), *PTh* is a thread process (accepts a wait statement in comparison to the method process), *PCTh* is a clocked thread process (sensitive to the clock event).

Definition 4.5 (*SystemC Program: `SC_Pg`*)

A SystemC program is a set $\langle L_{\text{SC_Mod}}, \text{SC_main} \rangle$, where $L_{\text{SC_Mod}}$ is a set of SystemC modules and *SC_main* is the main function in the program that performs the simulator initialization and contains the modules declarations.

4.1.2 Fixpoint Semantics

In this section, we define the semantics of the whole SystemC program, $\mathbb{W} \llbracket \text{SC_Pg} \rrbracket$, and the SystemC module, $\mathbb{M}_{\text{SC}} \llbracket m_{\text{sc}} \rrbracket$. Then, present the proofs (or proof sketches) of the soundness and completeness of $\mathbb{M}_{\text{SC}} \llbracket m_{\text{sc}} \rrbracket$.

Definition 4.6 (*Delta Delay: δ_d*)

The SystemC simulator considers two phases evaluate and update. The separation between these two phases is called delta delay.

Definition 4.7 (*SystemC Environment: `SC_Env`*)

The SystemC environment is the summation of the default C++ environment (*Env*) as defined in [20] and the signal environment (*Sig_Store*) specific to SystemC: $\text{SC_Env} = \text{Env} + \text{Sig_Env} = [\text{Var} \rightarrow \text{Addr}] + [\text{SC_Sig} \rightarrow \text{Addr}, \text{Addr}]$, where *Var* is a set of variables, *SC_Sig* is a set of SystemC signals and $\text{Addr} \subseteq \mathbb{N}$ is a set of addresses.

Definition 4.8 (*SystemC Store: SC_Store*)

The SystemC store is the summation of the default C++ store ($Store$) as defined in [20] and the signal store (Sig_Store): $SC_Store = Store + Sig_Store = [Addr \rightarrow Val] + [(Addr, Addr) \rightarrow (Val, Val)]$, where Val is a set of values such that $SC_Env \subseteq Val$.

Let $R_0 \in \mathcal{P}(SC_Env \times SC_Store)$ be a set of initial states, pc_{in} be the entry point of the main function and $\rightarrow \subseteq : (SC_Env \times SC_Store) \times (SC_Env \times SC_Store)$ be a transition relation.

Definition 4.9 (*Whole SystemC Program Semantics: $\mathbb{W} \llbracket SC_Pg \rrbracket$*)

Let $SC_Pg = \langle L_{SC_Mod}, SC_main \rangle$ be a SystemC program. Then, the semantics of SC_Pg , $\mathbb{W} \llbracket SC_Pg \rrbracket \in \mathcal{P}(SC_Env \times SC_Store) \rightarrow \mathcal{P}(T(SC_Env \times SC_Store))$ is

$$\mathbb{W} \llbracket SC_Pg \rrbracket(R_0) = \text{lfp}_{\emptyset}^{\subseteq} \lambda X. (R_0) \cup \{ \rho_0 \rightarrow \dots \rho_n \rightarrow \rho_{n+1} \mid \rho_{n+1} \in (SC_Env \times SC_Store) \wedge \{ \rho_0 \rightarrow \dots \rho_n \} \in X \wedge \rho_n \rightarrow \rho_{n+1} \}$$

Both definitions of the semantics of process declaration ($\mathbb{P}_R \llbracket SC_Pr \rrbracket$) and SystemC module constructor ($\mathbb{P}_{Ctr} \llbracket SC_Ctr \rrbracket$) are given in [14]. In contrast to the semantics definition of an OO object in [20], a SystemC method can be activated either by the default context or by the SystemC simulator through the sensitivity list of the process. A complete definition of the semantics of a SystemC module object ($\mathbb{O}_{SC} \llbracket o_sc \rrbracket$) through the definition of a transition function $\text{next}_{sc}(\sigma) = \text{next}(\sigma) \cup \text{next}_{sig}(\sigma)$, including both parts C++ related and SystemC specific functions, can be found in [14].

Definition 4.10 (*SystemC Module Semantics: $\mathbb{M}_{SC} \llbracket m_sc \rrbracket$*)

Let $m_sc = \langle DMem, Ports, Chan, Mth, SC_Ctr \rangle$ be a SystemC module, then its semantics $\mathbb{M}_{SC} \llbracket m_sc \rrbracket \in \mathcal{P}(T(\Sigma))$ is:

$$\mathbb{M}_{SC} \llbracket m_sc \rrbracket = \{ \mathbb{O}_{SC} \llbracket o_sc \rrbracket(v_{sc}, s_{sc}) \mid o_sc \text{ is an instance of } m_sc, v_{sc} \in D_{in}, s_{sc} \in SC_Store \}$$

Theorem 4.1 (*SystemC Module semantics in fixpoint*) Let

$$G_{sc} \langle S \rangle = \lambda T. \{ S_0 \langle v, s \rangle \mid \langle v, s \rangle \in S \} \cup \{ \sigma_0 \xrightarrow{l_0} \dots \xrightarrow{l_{n-1}} \sigma_n \xrightarrow{l'} \sigma' \mid \sigma_0 \xrightarrow{l_0} \dots \xrightarrow{l_{n-1}} \sigma_n \in T, \text{next}_{sc}(\sigma_n) \ni \langle \sigma', l' \rangle \}$$

Then $\mathbb{M}_{SC} \llbracket m_sc \rrbracket(v_{sc}, s_{sc}) = \text{lfp}_{\emptyset}^{\subseteq} G_{sc} \langle D_{in} \times Store \rangle$

Proof 1 (see [14])

The last step in the SystemC fixpoint semantics is to relate the module semantics to the whole SystemC program semantics. Hence, we consider an updated version of the function *abstract* (α°) as defined in [20]. The new function is upgraded to support the SystemC simulation semantics, environment and store. The complete definitions of α_{SC}° can be found in [14].

Theorem 4.2 (*Soundness of $\mathbb{M}_{SC}[\![m_{SC}]\!]$*) *Let M_{SC} be a whole SystemC program and let $m_{SC} \in M_{SC}$. Then $\forall R_0 \in SC_Env \times SC_Store$. $\forall \tau \in \mathcal{T}(SC_Env \times SC_Store)$.*

$$\tau \in \mathbb{W}[\![SC_Pg]\!](R_0) : \exists \tau' \in \mathbb{M}_{SC}[\![m_{SC}]\!]. \alpha_{SC}^\circ(\{\tau\}) = \{\tau'\}$$

Proof 2 (see [14])

Theorem 4.3 (*Completeness of $\mathbb{M}_{SC}[\![m_{SC}]\!]$*) *Let m_{SC} be a SystemC module. Then $\forall \tau \in \mathcal{T}(\Sigma)$. $\tau \in \mathbb{M}_{SC}[\![m_{SC}]\!]: \exists SC_P \in \langle L_{SC_Pg} \rangle$. $\exists \rho_0 \in SC_Env \times SC_Store$. $\exists o_{SC}$ instance of m_{SC} . $\exists \tau' \in \mathcal{T}(SC_Env \times SC_Store)$. $\tau' \in \mathbb{W}[\![\rho_0]\!] \wedge \alpha_{SC}^\circ(\{\tau'\}) = \{\tau\}$*

Proof 3 (see [14])

4.2 AsmL Fixpoint Semantics

4.2.1 Syntactical Domains

Definition 4.11 (*AsmL Class: AS_C*)

An AsmL class is a set $\langle AS_DMem, AS_Mth, AS_Ctr \rangle$, where AS_DMem is a set of the module data members, AS_Mth a set of methods (functions) definition and AS_Ctr is the module constructor.

One of the important features that we are going to use in AsmL corresponds to the methods pre-conditions (Boolean proposition verified before the execution of the method).

Definition 4.12 (*AsmL Method: AS_Mth*)

An AsmL method is a set $\langle AS_M, AS_Pre, AS_Pos, AS_Cst \rangle$, where AS_M is the method's core, AS_Pre is a set of pre-conditions, AS_Pos is a set of post-conditions and AS_Cst is a set of constraints.

Note that AS_Pre , AS_Pos and AS_Cst share the same structure. They are differentiated in the methods by using a specific keyword for each of them (e.g., *require* for pre-conditions).

Definition 4.13 (*AsmL Program: AS_Pg*)

An AsmL Program is a set $\langle L_{AS_C}, INIT \rangle$, where L_{AS_C} is a set of AsmL classes and $INIT$ is the main function in the program.

4.2.2 Fixpoint Semantics

Similar to the notion of delta delay (δ_d) of SystemC, AsmL considers two phases: *evaluate* and *update*. The program will be always running in the *evaluate* mode except if an update is requested. There are two types of updates, total and partial (usually performed using the `Step` instruction).

Definition 4.14 (*AsmL Environment: AS_Env*)

The AsmL Environment is a modified OO environment $AS_Env = [Var \rightarrow Addr, Addr]$, where Var is a set of variables and $Addr \subseteq \mathbb{N}$ is a set of addresses (two addresses store the current and new values of $v \in Var$).

Definition 4.15 (*AsmL Store: AS_Store*)

The AsmL store is $AS_Store = [(Addr, Addr) \rightarrow (Val, Val)]$, where Val is a set of values such that $AS_Env \subseteq Val$.

The whole AsmL program semantics ($\mathbb{W}_{AS} \llbracket AS_Pg \rrbracket$), method semantics ($\mathbb{M}_{AS} \llbracket \cdot \rrbracket$) and object semantics ($\mathbb{O}_{AS} \llbracket o_as \rrbracket$) through the definition of a transition function $next_{as}(\sigma)$ can be found in [15]. The AsmL class constructor can be defined according to the Definition 3.8 in [20].

Definition 4.16 (*AsmL Class Semantics: $\mathbb{C}_{AS} \llbracket c_as \rrbracket$*)

Let $c_as = \langle as_dmem, as_meth, as_ctr \rangle$ be an AsmL class, then its semantics $\mathbb{C}_{AS} \llbracket c_as \rrbracket \in \mathcal{P}(\mathcal{T}(\Sigma))$ is: $\mathbb{C}_{AS} \llbracket c_as \rrbracket = \{ \mathbb{O}_{AS} \llbracket o_as \rrbracket (v_as, s_as) \mid o_as \text{ is an instance of } c_as, v_as \in D_in, s_as \in SC_Store \}$

Theorem 4.4 (*AsmL Class semantics in fixpoint*) Let

$$H_{as} \langle S \rangle = \lambda T. \quad \{ S_o \langle v, s \rangle \mid \langle v, s \rangle \in S \} \cup \{ \sigma_0 \xrightarrow{l_0} \dots \xrightarrow{l_{n-1}} \sigma_n \xrightarrow{l'} \sigma' \mid \sigma_0 \xrightarrow{l_0} \dots \xrightarrow{l_{n-1}} \sigma_n \in T, next_{as}(\sigma_n) \ni \langle \sigma', l' \rangle \}$$

$$\text{Then } \mathbb{C}_{AS} \llbracket c_as \rrbracket (v_{as}, s_{as}) = lfp_{\emptyset}^{\subseteq} H_{as} \langle D_{in} \times Store \rangle$$

Proof 4 (see [15])

The function α_{AS}° is an updated version of the function *abstract* (α°) defined in [20]. The complete definition of α_{AS}° is given in [15].

Theorem 4.5 (*Soundness of $\mathbb{C}_{AS} \llbracket c_as \rrbracket$*) Let P_{AS} be a whole AsmL program and let $c_{AS} \in \mathcal{C}_{AS}$. Then $\forall R_0 \in AS_Env \times AS_Store. \forall \tau \in \mathcal{T}(AS_Env \times AS_Store).$

$$\tau \in \mathbb{W} \llbracket AS_Pg \rrbracket (R_0) : \exists \tau' \in \mathbb{C}_{AS} \llbracket c_{AS} \rrbracket. \alpha_{AS}^\circ(\{\tau\}) = \{\tau'\}$$

Proof 5 (see [15])

Theorem 4.6 (Completeness of $\mathbb{C}_{AS}[\![\]\!]$) Let c_{AS} be a AsmL class. Then

$$\forall \tau \in \mathcal{T}(\Sigma). \quad \tau \in \mathbb{C}_{SC}[\![c_{SC}]\!] : \exists AS_P \in \langle L_{AS_Pg} \rangle. \exists \rho_0 \in AS_Env \times AS_Store. \exists o_{AS} \text{ instance of } c_{AS}. \exists \tau' \in \mathcal{T}(AS_Env \times AS_Store). \tau' \in \mathbb{W}[\![\rho_0]\!] \wedge \alpha_{AS}^o(\{\tau'\}) = \{\tau\}$$

Proof 6 (see [15])

4.3 Program Transformation

The equivalence in behavior, with respect to an observation α_o , between the source SystemC program and the target AsmL program is required to ensure the soundness of any verification result at the AsmL level. Our objective is to define a relation between the SystemC processes active for certain delta cycle and the set of methods allowed to be executed in the AsmL model. Hence, we will map every thread (method, sensitivity list) in the SystemC design to a method (method core, pre-conditions) in the AsmL model.

The SystemC observation function needs to see all the active processes at the beginning of a delta-cycle by checking for the end of the update phase.

Definition 4.17 (SystemC observation function: α_o^{SC})

Let $SC_Pg = \langle L_{SC_Mod}, SC_main \rangle$ be a SystemC program, the observation function $\alpha_o^{SC} \in \mathcal{P}(SC_Env \times SC_Store) \rightarrow \mathcal{P}(\mathcal{T}(SC_Env \times SC_Store))$ is

$$\alpha_o^{SC}[\![SC_Pg]\!](R_0) = \text{Ifp}_{\emptyset}^{\subseteq} \lambda X. R_0 \cup \{ \tilde{\rho}_0 \rightarrow \dots \tilde{\rho}_n \mid \forall \tilde{\rho}_i \in (SC_Env \times SC_Store) \exists \{ \rho_0^i \rightarrow \dots \rho_m^i \} \in X \wedge \rho_m^i \rightarrow \tilde{\rho}_i \wedge \{ m_sc \text{ in } \mathbb{M}_{SC} \mid \exists o_sc \in \mathbb{M}_{SC}. o_sc(\rho_m^i()) \neq \{\epsilon\} \} = \emptyset \}$$

In the previous definition, α_o^{SC} is only tracing the initial states of a simulation cycle. For instance, the third condition ensures that the list of process ready to run is empty. Similarly, we define an observation function α_o^{AS} for an AsmL program.

Definition 4.18 (AsmL observation function: α_o^{AS})

Let $AS_Pg = \langle L_{AS_C}, INIT \rangle$ be an AsmL program, the observation function $\alpha_o^{AS} \in \mathcal{P}(AS_Env \times AS_Store) \rightarrow \mathcal{P}(\mathcal{T}(AS_Env \times AS_Store))$ is

$$\alpha_o^{AS}[\![AS_Pg]\!](R_0) = \text{Ifp}_{\emptyset}^{\subseteq} \lambda X. (R_0) \cup \{ \tilde{\rho}_0 \rightarrow \dots \tilde{\rho}_n \mid \forall \tilde{\rho}_i \in (SC_Env \times AS_Store) \exists \{ \rho_0^i \rightarrow \dots \rho_m^i \} \in X \wedge \rho_m^i \rightarrow \tilde{\rho}_i \wedge \{ m_as \text{ in } \mathbb{C}_{AS} \mid \exists o_as \in \mathbb{C}_{AS}. o_as(\rho_m^i()) \neq \{\epsilon\} \} = \emptyset \}$$

Next, we define the notion of equivalence between the two observations. Although, SystemC and AsmL have different environment and store structures, it is possible to ensure that they contain the same information.

Definition 4.19 (Equivalence w.r.t. $\alpha_o: \equiv_{\alpha_o}$)

Let SC_Pg be a SystemC program, V_sc a set of its variables, AS_Pg be an AsmL program and $Dout_as$ a set of its output variables.

$prog_sc \equiv_{\alpha_o} prog_as$ if

$\forall R_0^{SC}$ set of initial states of SC_Pg . $\forall R_0^{AS}$ set of initial states of AS_Pg .

$\forall \tilde{\rho} \in \{\tilde{\rho}_0 \rightarrow \dots \rightarrow \tilde{\rho}_n\} \in \alpha_o^{SC} \llbracket SC_Pg \rrbracket (R_0^{SC})$.

$\exists \hat{\rho} \in \{\hat{\rho}_0 \rightarrow \dots \rightarrow \hat{\rho}_n\} \in \alpha_o^{AS} \llbracket AS_Pg \rrbracket (R_0^{AS}) \mid \forall vsc \in V_sc. \exists vas \in V_as$ such that
 if $vsc \in SC_Sig$ then $(\tilde{\rho}(vsc) = (v11, v12)) \wedge (\hat{\rho}(vas) = (v11, v12))$
 if $vsc \in AS_DMem$ then $(\tilde{\rho}(vsc) = v11) \wedge (\hat{\rho}(vas) = (v11, v11))$

The observation function ensures that the AsmL program is mimicking the *evaluate* and *update* phases (same length n of the ρ sets). The first if condition takes care of the SystemC signals while the second one concerns basic C++ variables.

Theorem 4.7 (Existence of transformed AsmL program w.r.t. α_o^{SC}) Let SC_Pg be a whole SystemC program, SC_Din a set of inputs and SC_Dout a set of outputs. Then

$\exists AS_Pg$, an AsmL program, such that $SC_Pg \equiv_{\alpha_o} AS_Pg$

Proof 7 (see [16])

Theorem 4.8 (Existence of transformed SystemC program w.r.t. α_o^A) Let AS_Pg be a whole AsmL program, AS_Din a set of inputs and AS_Dout a set of outputs. Then

$\exists SC_Pg$, a SystemC program, such that $AS_Pg \equiv_{\alpha_o} SC_Pg$

Proof 8 (see [16])

Theorem 4.9 (Soundness of the transformations) Let SC_Pg be a whole SystemC program and let AS_Pg be a whole AsmL program. Then

$SC_Pg \equiv_{\alpha_o} AS_Pg$:

$\forall Prop(V_sc, \tilde{\rho}) \mid \tilde{\rho} \in \alpha_o^{SC} \llbracket SC_Pg \rrbracket. SC_Pg \vdash Prop(V_sc, \tilde{\rho})$

$: AS_Pg \vdash Prop(V_as, \hat{\rho}) \mid \hat{\rho} \in \alpha_o^{AS} \llbracket AS_Pg \rrbracket$.

where $Prop$ is a program's property, V_sc is a set of variables of the SystemC program, V_as are their corresponding variables in the AsmL program.

Proof 9 (see [16])

5 Experimental Results

In order to illustrate the proposed design and verification methodology, we considered two models: (1) PCI (Peripheral Component Interconnect) [8] local bus standard; and (2) an extension of the Master/Slave Bus structure provided by the SystemC distribution [25]. Both models include certain 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.

Additional case studies can be found in [11] and [10]. In particular in [10], we compared the performances of our approach to the RuleBase [19] model checker on a look-aside interface [24]. Experiments showed the performance of our approach to handle cases that classical HDL model checkers cannot support (due to state explosion problem).

5.1 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 (if more than one are connected to the bus). Two modes are supported by the bus: (1) *Blocking Mode*: data is moved through the bus in a burst-mode; and (2) *Non-Blocking Mode*: the master reads or writes a single data word.

5.2 Model Checking

For model checking we consider, 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 CPU time 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.

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

Number of		Model Checking			Simulation
Masters	Slaves	CPU	Number of FSM		Avg. Exc. per cycle δ (10^{-9} s)
		Time (s)	Nodes	Trans.	
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

5.3 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 (δ 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.

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

Number of			Model Checking			Simulation
Slaves	Masters		CPU	Number of FSM		Avg. Exc. per cycle δ (10^{-9} s)
	B	NB	Time (s)	Nodes	Trans.	
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

5.4 Assertions Coverage Enhancement

In order to evaluate the proposed genetic algorithm, we considered the Master/Slave bus structure model and a set of 10 assertions¹. Table 3 compares the assertion coverage results obtained: (a) with a blind random generation; (b) in the initialization phase of the genetic algorithm (GA), i.e., just after the first DNA was generated from the static analysis phase; and (c) after 35 generations of the GA. We used 10^9 simulation cycles for every generation. The coverage is measuring the percentage of test vectors that updated at least one of the variables of the assertion. We clearly notice that the static analysis phase already offers a better initial state than starting with totally random generation.

Figure 8 gives more details about the evolution of the algorithm for the three

¹For more details, we refer the reader to <http://hvg.ece.concordia.ca/Research/SoC/GeneticAlgo/>.

Table 3: Assertions' coverage analysis

Assertion	Random (%)	Init. of GA(%)	GA after 35 iter.(%)
A1	10	34	92
A2	8	42	93
A3	12	32	85
A4	11	37	89
A5	14	41	87
A6	16	46	91
A7	10	41	94
A8	17	33	83
A9	16	31	82
A10	14	45	97

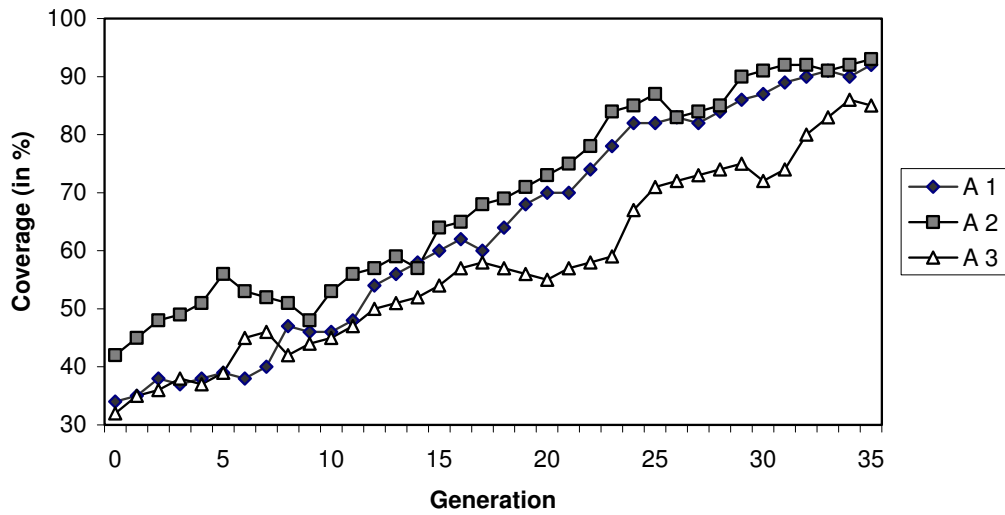


Figure 8: Assertion coverage evolution as function of the Population Generation.

assertions (A1, A2 and A3). Typically, a genetic algorithm makes relatively quick progress in the beginning stages of evolution. We noted that there exist some

phases, where the algorithm hits local maxima before mutating further, which improves its performance. We even noticed that the coverage sometimes decreases slowly from generation to generation (for e.g., generation 20 for A3). This is due to the fact that the evaluation of the assertion is based on weighted random generation. In other terms, since the number of tests is finite, a generator may have two different coverage results for two different test trials.

6 Related Work

Related work to ours concerns mainly: (1) defining system level design methodologies using SystemC; (2) writing the formal semantics of PSL, SystemC and AsmL; (3) using genetic algorithms to enhance assertions coverage; and (4) performing program transformation from SystemC to AsmL and vice-versa.

Several proposals for system level design, in particular [28], 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 an in-design verification solution.

Genetic algorithms have already been used for a broad range of applications. The most related work to ours is the one of Godefroid *et al.* [5], which in contrast to other approaches, addressed in particular the exploration of large state spaces of concurrent reactive systems as defined for model checking. Nevertheless, this work was restricted to simple Boolean assertions and was based on BDDs which is not suitable for high level languages like SystemC. We added to [5] a static analysis phase before applying the genetic algorithm. We also considered a chromosome-encoding based on weighted probability over the space of the possible values of the program variables. We are not aware of any other work where genetic algorithms have been combined with static code analysis to optimize test vector generators in order to improve assertions coverage for SystemC.

In [6], 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 [7]. 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, he does not offer any approach to re-use the PSL properties as assertion (a very

important feature in PSL).

Several approaches have been used to write the SystemC semantics (e.g., using ASM is [22]). Denotational semantics [23] is found to be most effective since objects can be expressed as fixpoints on suitable domains. Salem in [27] proposed a denotational semantics for SystemC; but, his proposal was very shallow, missing to relate the semantics of the whole SystemC program to the semantics of its classes. Therefore, in order to construct a transformation relation between SystemC and AsmL and to prove its soundness, we defined, in this report, our own SystemC denotational semantics.

Regarding, the program transformation, the work of Patrick and Radhia Cousot in [3] is the essence for any program transformation using abstract interpretation. We used a projection of that generic approach, described in Section 3.9 of [3] on a SystemC subject program and an AsmL transformed program can be used to perform the soundness of a transformation and also to construct it. In both cases, we defined the syntax, semantics and observation functions for both AsmL and SystemC.

7 Conclusion

In this report, we presented a methodology to design and verify SystemC transactional models starting from a UML system specification and integrating an intermediate layer using the AsmL language. We proposed to upgrade the UML sequence diagram in order to capture transaction related system properties. Then, both the design and its properties are modeled in AsmL 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 AsmL to its implementation in SystemC.

In order to efficiently verify assertions in SystemC, we further apply a static code analysis technique based on abstract interpretation. This phase generates an abstracted version of the initial design modeled as a hypergraph that helps defining the dependency between the system inputs and the assertion's variables, as well as restricting the possible values of the inputs to certain ranges that may update the assertion. Although experiments showed that this approach improves the assertion's coverage, we proposed to use a genetic algorithm that optimizes the probability distribution of the inputs over the space of their possible values.

We also presented the fixpoint semantics of the SystemC library including, in

particular, the semantics of a SystemC Module that we proved to be sound and complete w.r.t. a trace semantics of a SystemC program. We provided also the semantics of a subset of AsmL and we proved the soundness and completeness of an AsmL class w.r.t. to a trace semantics of the AsmL program. Then, we proved the existence, for every SystemC program, of an AsmL program having similar behavior w.r.t. an observation function that we set to consider the traces of the system just after the update phase of the SystemC simulator. We have used this SystemC to AsmL transformation to reduce the complexity of SystemC models and enabled their formal verification using model checking and theorem proving approaches used with AsmL and ASM languages in general.

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. Our genetic algorithm showed an improvement of the assertions coverage by a factor of eight in comparison to the random case. As future work in this direction, we target to optimize the genetic algorithm to improve various coverage metrics.

References

- [1] Accellera Organization. Accellera property specification language reference manual, version 1.01. <http://www.accellera.org>, 2004.
- [2] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.
- [3] P. Cousot and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *Proc. Symposium on Principles of Programming Languages*, pages 178–190, Portland, Oregon, January 2002.
- [4] R. Damasevicius and V. Stukys. Application of UML for hardware design based on design process model. In *Proc. Asia South Pacific Design Automation Conference*, pages 244–249, 2004.
- [5] P. Godefroid and S. Khurshid. Exploring very large state spaces using genetic algorithms. In *Proc. Tools and Algorithms for Construction and Analysis of Systems*, 2002.
- [6] M. Gordon. Validating the PSL/Sugar semantics using automated reasoning. In *Proc. Formal Aspects of Computing*, pages 306–421, 2003.

- [7] M. Gordon and T. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge Univ. Press, 1993.
- [8] PCI Special Interest Group. <http://www.pcisig.com/>, 2004.
- [9] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
- [10] A. Habibi, A.I. Ahmed, O. Ait-Mohamed, and S. Tahar. On the design and verification of the look-aside interface. In *Proc. Design Automation and Test in Europe*, Munich, Germany, (to appear).
- [11] A. Habibi, A. Gawanmeh, and S. Tahar. Assertion based verification of PSL for SystemC designs. In *Proc. IEEE International Symposium on System-on-Chip (SOC'04)*, Tampere, Finland, November 2004.
- [12] A. Habibi and S. Tahar. Design for verification of SystemC transaction level models. In *Proc. Design Automation and Test in Europe*, Munich, Germany, (to appear).
- [13] A. Habibi and S. Tahar. Towards an efficient assertion based verification of SystemC designs. In *Proc. High Level Design Validation and Test Workshop*, Sonoma Valley, CL, USA, November 2004.
- [14] A. Habibi and S. Tahar. SystemC fixpoint semantics. Technical report, ECE, Concordia University, December 2004 (www.ece.concordia.ca/~habibi/tehrp/TR0401/).
- [15] A. Habibi and S. Tahar. AsmL fixpoint semantics. Technical report, ECE, Concordia University, December 2004 (www.ece.concordia.ca/~habibi/tehrp/TR0402/).
- [16] A. Habibi and S. Tahar. Constructing sound SystemC/AsmL and AsmL/SystemC transformations. Technical report, ECE, Concordia University, December 2004 (www.ece.concordia.ca/~habibi/tehrp/TR0403/).
- [17] A. Habibi and S. Tahar. Abstract interpretation of SystemC designs. Technical report, Department of ECE, Concordia University, June 2004 (www.ece.concordia.ca/~habibi/tehrp/TR0404/).
- [18] J. Holland. *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, 1975.

- [19] IBM Haifa Research Laboratories. *RuleBase Formal Verification Tool (Version 1.5). Users Guide*. May 2003.
- [20] F. Logozzo. *Analyse Statique Modulaire de Langages a Objets*. PhD thesis, Ecole Polytechnique, Paris, France, June 2004.
- [21] Microsoft Corp. AsmL for Microsoft .NET Framework. research.microsoft.com.
- [22] W. Müller, J. Ruf, and W. Rosenstiel. *SystemC Methodologies and Applications*. Kluwer Academic Pub., 2003.
- [23] P. D. Mosses. *Denotational semantics*, volume B of *Handbook of Theoretical Computer Science*, chapter 11, pages 575–631. Elsevier Science B.V., 1990.
- [24] Network Processing Forum. *Look-Aside (LA-1) Interface, Implementation Agreement, Revision 1.1*. Kluwer Academic Publishers, April 15, 2004.
- [25] Open SystemC Initiative. <http://www.systemc.org>, 2004.
- [26] H. Rudin. Protocol development success stories: Part i. In *Proc. International Symposium on Protocol Specification, Testing, and Verification*, Florida, USA, June 1992.
- [27] A. Salem. Formal semantics of synchronous SystemC. In *Proc. Design, Automation and Test in Europe Conference*, pages 376–381, Munich, Germany, March 2003.
- [28] C. Schulz-Key, M. Winterholer, T. Schweizer, T. Kuhn, and W. Rosenstiel. Object-oriented modeling and synthesis of SystemC specifications. In *Proc. Conference on Asia South Pacific design automation*, pages 238–243. IEEE Press, 2004.
- [29] F. Vederine. *Analyses totales de programmes par interpretation abstraite*. PhD thesis, Ecole Polytechnique, Paris, France, 2000.