An Approach for the Verification of SystemC Designs Using AsmL

Ali Habibi and Sofiène Tahar

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

Abstract. The spectacular advancement in microelectronics resulted in the creation of new system level design languages, such as SystemC, which put fourth new design and verification challenges. In this paper, we present an approach verifying SystemC designs using model checking and assertion based verification. Such verification is enabled through two transformations from SystemC to AsmL (the Abstract State Machines Language) and vice-versa. The soundness of these transformations, proved using abstract interpretation, guarantees the correctness of the model checking results and the validity of the generated assertion monitors (to be checked by simulation). We illustrate our approach on the SystemC/AsmL modeling and verification of the widely used Accelerated Graphics Port (AGP) standard. The verified AGP model can be either refined to implement an AGP core or used to validate existent compatible device.

1 Introduction

SystemC [18] is an object-oriented system level language for embedded systems design and verification. It is expected to make a stronger effect in the area of architecture, co-design and integration of hardware and software. The SystemC library is composed of a set of classes and a simulation kernel extending C++ to enable the modeling of complex systems at a higher level of abstraction than state-of-the-art HDLs. Nevertheless, except for small models, the verification of SystemC designs is a serious bottleneck in the system design flow. While simulation is the mostly 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 of SystemC is not feasible due to the complexity of this library. 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. The soundness of ABV relies, in particular, on the correctness of the generation of the SystemC monitor from the property.

In order to enable the model checking of a SystemC design, we translate it to an intermediate representation in AsmL [16]. This latter is an object-oriented abstract state machines (ASM) [2] description language providing features to

D.A. Peled and Y.-K. Tsay (Eds.): ATVA 2005, LNCS 3707, pp. 69–83, 2005.

capture the behavioral semantics of programming and modeling languages where systems are modeled at a high level of abstraction allowing easier validation and verification operations.

The AsmL language is integrated with Microsoft's software development environment and integrated with the Asmlt tool [16] offering a reachability algorithm, able to generate an FSM of the model that can be adapted to perform model checking. When a state explosion happens the design properties are translated to SystemC assertion monitors and verified by simulation. This is made possible through the embedding of the property specification language (PSL [1]) in the same formalism.

The soundness of our approach is established using abstract interpretation by proving the correctness of both transformations: (1) from the original SystemC design to its AsmL representation; and (2) from the PSL property, in AsmL, to the generated monitor, in SystemC.

To illustrate our approach, we considered the AGP bus [14] that was, as far as we know, only verified by simulation due to its complexity and very large state space. We will show that our technique combined with the abstraction features of AsmL allows, using an inductive proof, the model checking of a set of properties on the bus. These properties are also translated to a SystemC monitor that can be used as a separate Intellectual Property (IP) to validate AGP compatible devices.

The rest of this paper is organized as follows: Section 2 discusses related work. Section 3 presents our verification approach. Section 4 contains the proofs of the transformation from SystemC to AsmL. Section 5 describes the application of the proposed methodology for the case of an AGP bus modeled in SystemC. Finally, Section 6 concludes the paper.

2 Related Work

Related work to ours concerns both finite-state verification and assertion based verification. Concerning the first issue, we cite in particular the Bandera [5] project that aims at interfacing Java code to model checking tools like SMV [3] and SPIN [13] 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. Besides, using SMV as an internal model checking tool is a big handicap for Bandera to handle large state space systems. We are not aware of any related work using a sound syntactical transformation from SystemC to AsmL and vice-versa to perform either model checking or ABV.

In [7] an approach is presented to add assertion checkers to SystemC. This previous work is different from our methodology mainly in two aspects: (1) The properties in [7] are restricted to the notation of property checker from Infineon Technologies AG then translated to synthesizable SystemC instructions while we consider any PSL property; and (2) SystemC is considered in [7] as a low level

HDL language while in this paper we do not put any restriction on any subset of SystemC.

In [19], [12] and [10] several approaches were proposed to verify, respectively, a PCI bus monitor in Verilog, a PCI bus model and Look-aside interface [17] (both in SystemC). In [19], the bus was implemented in Verilog with all the properties embedded as part of the code which makes its modification or upgrade a very complex task. Besides, the verified Verilog model includes only two agents (one master and one slave), which does not allow the verification of the properties related to the bus arbitration, for example, and radically reduces the designs state space.

In both [12] and [10] a top-down approach was used where the verification was integrated as part of the design process and AsmL models were first designed and verified then translated to SystemC. In this paper, we consider a bottom-up approach where starting from an existent AGP IP in SystemC we generate internally the AsmL model and verify the system property at the ASM level. Besides, the designs in [19], [10] and [17] were relatively small in comparison to AGP, with a width of 256 for data read, data write and command queues has a minimum of $2^{256\times32}$ states. Furthermore, AGP includes a number of additional features making its verification a non-trivial task, such as pipelining. Hence, direct model checking of AGP properties is with no doubt impossible due to the state-explosion problem. The verification technique proposed in this paper takes advantage of the high level of abstraction offered by AsmL which enables both data abstraction and proofs by induction.

3 Verification Methodology

AsmL [9] is one of the very latest languages developed for Abstract State Machines (ASM) [8]. It is supported by a tester (Asmlt) that can be used to generate FSMs and test cases. It supports object-oriented modeling at higher level of abstraction in comparison to C++ and Java. In our verification methodology (Figure 1) we perform the model checking of SystemC by translating the original design to an intermediate representation that omits all the details of the SystemC simulator. The target (or transformed) program is modeled in AsmL to be cross-produced with the system properties that will be verified over the whole system's state space. To model the properties, we used the PSL [1] standard. PSL properties are embedded in the design as external monitors; hence, they can be used as stand-alone IP block(s) to validate other devices, either at the AsmL level by model checking or at the SystemC level by assertion based verification.

3.1 Model Checking

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 properties evaluation in every state, which enables checking its correctness on-the-fly while

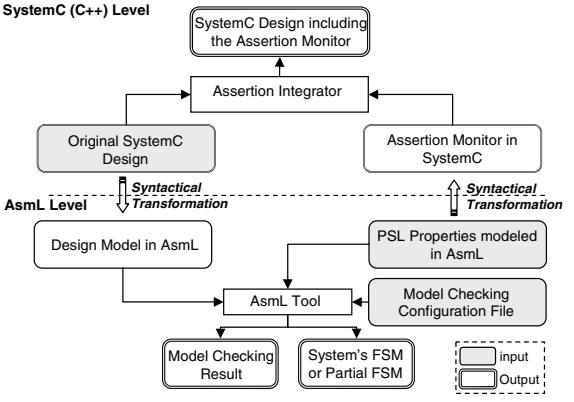

Fig. 1. Verification Methodology

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 scenario for a counter-example.

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. The main layers include the Boolean layer, the temporal layer and the verification layer [1].

We encapsulate sequences in the verification unit as an assertion, which is embedded in the design. Given a set of Boolean items $x_1, x_2, \ldots, x_n$, and $y_1, y_2, \ldots, y_m$ belonging to the Boolean layer, and the sequences, S_1 and S_2 belonging to the temporal layer, we can define: $S_1 = \{x_1, x_2, \ldots, x_n\}$, and $S_2 = \{y_1, y_2, \ldots, y_m\}$ and then use assertions to check any PSL operation between S_1 and S_2 such as $S_1\ OP\ S_2$, where OP is a PSL operator (e.g., implication (:), or equivalence ($\Leftrightarrow$)). The assertion is built as follows:

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

Each 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} (stating, 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 $P_{value} = false$. The previous condition is a filter for the FSM generation algorithm stopping the generation when an error is detected. In this case, the generated portion of the state machine can be used to identify the problem through a scenario of a counter-example. For multiple properties, the filter is set as the

conjunction of all the conditions for the separate properties. This technique minimizes radically the number of state variables (the FSM size and its generation time). A successful verification process results in the generation of the system's FSM (according to the configuration file constraints). This approach may seem to be based on an ad-hoc model checking algorithm while more advanced techniques and approaches have been used in tools like SMV and VIS. We believe there are many reasons that make our approach more efficient, in particular:
(1) It is impossible to use these tools with AsmL considering the OO nature of the language. Therefore, a translation to the language supported by the tool (mostly a very low HDL) is mandatory. This operation will prohibit using some advanced features AsmL offers (e.g., data abstraction, etc.)
(2) Generating the counter-example as an FSM provides a complete path of the error starting from the entry point to the state where the error took place [6].
(3) The configuration of the FSM generation algorithm can be set by the user in order to stress the verification only in some particular portions of the state space (through restricting some variables to have certain range for example) [6].

3.2 Assertion Based Verification

The proposed methodology to integrate and verify PSL assertions for SystemC designs is given in Figure 2. It consists of the following three main steps:

(1) Updating the SystemC design in order to interface it with the assertion monitor.
(2) Generating the assertion as a C# code from its ASM description.
(3) Integrating the C# assertion in the SystemC design.

The assertion under verification is a PSL property embedded in AsmL as a read-only separate module. In order to guarantee that we are verifying the same property specified in AsmL as the corresponding SystemC model, we need to: (1) prove the correctness of the transformation from AsmL to SystemC; and (2) connect the assertion monitor correctly to the original SystemC design. The

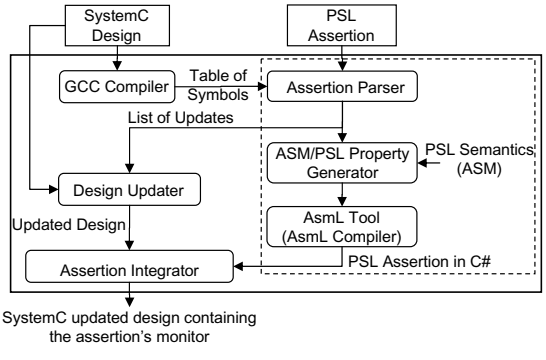

Fig. 2. Assertion Based Verification Approach

second step requires updating the SystemC design to interface to the assertion and integrating the assertion in the design. For instance, we validate the assertion syntactically by generating the list of the variables involved. Then, we perform a type check to make sure the variables are well instantiated in the SystemC design. For instance, the signals (variables) that are used in the assertion must be seen as external signals so that they can be input to the assertion monitor. Hence, we modify the SystemC design to make the required variables visible to the monitor. Once the design is updated, we add the required instantiation of the assertion to bind it to the existing SystemC design modules. The assertion monitor, acting as part of the design, can do the following: (1) stop the simulation when the assertion is fired; (2) write a report about the assertion status and all its variables; and (3) send a warning signal to other modules (if required).

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

The work of Patrick and Radhia Cousot in [4] 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 [4]. Figure 3 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.

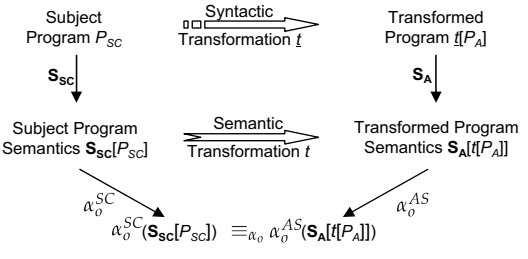

Fig. 3. Online Program Transformation

4.1 SystemC Fixpoint Semantics

Syntactical Domains. SystemC has 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 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 (function) definition and* `SC_Ctr` *the module constructor.*

Definition 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 3. *(SystemC Constructor: SC_Ctr)*
A SystemC Constructor is a set $\langle$`Name, Init, SC_Pr, 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. *(SystemC Process: SC_Pr)*
A SystemC process is a set $\langle$`PMth, PTh, PCTh`$\rangle$*, where* `PMth` *is a method process (defined as a set* $\langle$`Mth, 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 5. *(SystemC Program: SC_Pg)*
A SystemC program is a set $\langle L_{\texttt{SC\_Mod}}, \texttt{SC\_main}\rangle$*, where* $L_{\texttt{SC\_Mod}}$ *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.*

Fixpoint Semantics. In this section, we define the semantics of the whole SystemC program, $\mathbb{W}$ $[\![\texttt{SC\_Pg}]\!]$, and the SystemC module, $\mathbb{M}_{SC}[\![\texttt{m\_sc}]\!]$. Then, present the proofs (or proof sketches) of the soundness and completeness of $\mathbb{M}_{SC}[\![\texttt{m\_sc}]\!]$.

Definition 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 7. *(SystemC Environment: SC_Env)*
The SystemC environment is the summation of the default C++ environment (`Env`*) as defined in [15] and the signal environment (*`Sig_Store`*) specific to SystemC:* `SC_Store = Env + Sig_Env = [Var → Addr]+ [SC_Sig → (Addr,Addr)]`*, where* `Var` *is a set of variables,* `SC_Sig` *is a set of SystemC signals and* `Addr` $\subseteq \mathbb{N}$ *is a set of addresses.*

Definition 8. *(SystemC Store: SC_Store)*
The SystemC store is the summation of the default C++ store (Store) as defined in [15] and the signal store (Sig_Store): SC_Store = Store + Sig_Store = [Addr → Val]+ [(Addr, Addr) → (Val,Val)], where Val is a set of values such that SC_Env ⊆ Val.

Let $\mathtt{R}_0 \in \mathcal{P}(\mathtt{SC\_Env}\times\mathtt{SC\_Store})$ be a set of initial states, $\mathtt{pc_{in}}$ be the entry point of the main function and $\rightarrow\subseteq$: $(\mathtt{SC\_Env}\times\mathtt{SC\_Store})\times(\mathtt{SC\_Env}\times\mathtt{SC\_Store})$ be a transition relation.

Definition 9. *(Whole SystemC Program Semantics: $\mathbb{W}$ SC_Pg)*
Let SC_Pg = $\langle L_{SC\_Mod}, SC\_main\rangle$ be a SystemC program. Then, the semantics of SC_Pg, $\mathbb{W}$ SC_Pg $\in \mathcal{P}(SC\_Env\times SC\_Store) \rightarrow \mathcal{P}(\mathcal{T}(SC\_Env\times SC\_Store))$ is:

$$\mathbb{W}SC\_Pg(R_0) = lfp^{\subseteq}_{\emptyset}\lambda X.\,(R_0) \cup \{\rho_0 \rightarrow \ldots \rho_n \rightarrow \rho_{n+1} |\ \rho_{n+1} \in (SC\_Env\times SC\_Store) \wedge \{\rho_0 \rightarrow \ldots \rho_n\} \in X \wedge \rho_n \rightarrow \rho_{n+1}\}$$

Both definitions of the semantics of process declaration ($\mathbb{P}_R$ $[\![\mathtt{SC\_Pr}]\!]$) and SystemC module constructor ($\mathbb{P}_{Ctr}$ $[\![\mathtt{SC\_Ctr}]\!]$) are given in [11]. In contrast to the semantics definition of an OO object in [15], 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 ($\mathbb{O}_{SC}[\![\mathtt{o\_sc}]\!]$) through the definition of a transition function $\mathtt{next_{sc}}(\sigma)$=next($\sigma$) $\bigcup$ $\mathtt{next_{sig}}(\sigma)$, including both parts C++ related and SystemC specific functions, can be found in [11].

Definition 10. *(SystemC Module Semantics: $\mathbb{M}_{SC}$m_sc))*
Let m_sc = $\langle$DMem, Ports, Chan, Mth, SC_Ctr$\rangle$ be a SystemC module, then its semantics $\mathbb{M}_{SC}$m_sc) $\in \mathcal{P}(\mathcal{T}(\Sigma))$ is:

$$\mathbb{M}_{SC}m\_sc = \{\mathbb{O}_{SC}o\_sc(v_{sc}, s_{sc}) \mid o\_sc \text{ is an instance of } m\_sc,\ v\_sc \in D\_in,\ s\_sc \in SC\_Store\}$$

Theorem 1. *(SystemC Module semantics in fixpoint)* [1] *Let*

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

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

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 [15]. The new function is upgraded to support the SystemC simulation semantics, environment and store. The complete definitions of $\alpha\_\mathtt{SC}^\circ$ can be found in [11].

Theorem 2. *(Soundness of $\mathbb{M}_{SC}$m_sc) Let M_{SC} be a whole SystemC program and let $m_{SC} \in M_{SC}$. Then:*

$$\forall\, R_0 \in SC\_Env\times SC\_Store.\ \forall\, \tau \in \mathcal{T}(SC\_Env\times SC\_Store).\ \tau \in \mathbb{W}SC\_Pg(R_0) : \exists\tau' \in \mathbb{M}_{SC}m_{SC}\ .\ \alpha\_SC^\circ(\{\tau\}) = \{\tau'\}$$

[1] The proofs of the theorems presented in this paper are available in [11].

Theorem 3. *(Completeness of $\mathbb{M}_{SC}$) 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*). $\tau' \in \mathbb{W}\rho_0 \wedge\ \alpha\_SC^{\circ}(\{\tau'\}) = \{\tau\}$

4.2 AsmL Fixpoint Semantics

Syntactical Domains.

Definition 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 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 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.*

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.

Definition 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 as set of addresses (two addresses store the current and new values of* $v \in$ `Var`*).*

Definition 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 ($\mathbb{W}_{AS}$ $[\![$`AS_Pg`$]\!]$), method semantics ($\mathbb{M}_{AS}$ $[\![.]\!]$) and object semantics ($\mathbb{O}_{AS}[\![$`o_AS`$]\!]$) through the definition of a transition function $\texttt{next}_{\texttt{AS}}(\sigma)$ can be found in [11]. The AsmL class constructor can be defined according to the Definition 3.8 in [15].

Definition 16. *(AsmL Class Semantics: $\mathbb{C}_{AS}$ c_as)*
Let $\texttt{c\_as} = \langle \texttt{as\_dmem}, \texttt{as\_mth}, \texttt{as\_ctr} \rangle$ *be an AsmL class, then its semantics* $\mathbb{C}_{AS}\,\texttt{c\_as}) \in \mathcal{P}(\mathcal{T}(\Sigma))$ *is:* $\mathbb{C}_{as}\,\texttt{c\_as} = \{\mathbb{O}_{AS}\,\texttt{o\_as}(\texttt{v\_as}, \texttt{s\_as}) \mid \texttt{o\_as}$ *is an instance of* $\texttt{c\_as}, \texttt{v\_as} \in \texttt{D\_in}, \texttt{s\_as} \in \texttt{SC\_Store}\}$

Theorem 4. *(AsmL Class semantics in fixpoint) Let*

$$H_{as}\langle S\rangle = \lambda T.\ \{\texttt{S\_O}\langle v, s\rangle \mid \langle v, s\rangle \in S\ \} \cup \{\sigma_0 \xrightarrow{l_0} \ldots \xrightarrow{l_{n-1}} \sigma_n \xrightarrow{l'} \sigma' \mid \sigma_0 \xrightarrow{l_0} \ldots \xrightarrow{l_{n-1}} \sigma_n \in T,\ \texttt{next}_{\texttt{AS}}(\sigma_n) \ni \langle \sigma', l'\rangle\}$$

Then $\mathbb{C}_{AS}\,\texttt{c\_as}(v_{as}, s_{as}) = lfp^{\subseteq}_{\emptyset}\ H_{as}\langle\ \texttt{D\_in} \times \texttt{Store}\rangle$

The function $\alpha\_\texttt{AS}^{\circ}$ is an updated version of the function *abstract* (α°) defined in [15]. The complete definition of $\alpha\_\texttt{AS}^{\circ}$ is given in [11].

Theorem 5. *(Soundness of* $\mathbb{C}_{AS}\,\texttt{c\_as}$*) Let* P_{AS} *be a whole AsmL program and let* $\texttt{c}_{AS} \in \mathcal{C}_{AS}$*. Then* $\forall\ \texttt{R}_0 \in \texttt{AS\_Env} \times \texttt{AS\_Store}.\ \forall\ \tau \in \mathcal{T}(\texttt{AS\_Env} \times \texttt{AS\_Store}).\ \tau \in \mathbb{W}\texttt{AS\_Pg}(\texttt{R}_0) : \exists \tau' \in \mathbb{C}_{AS}\,\texttt{c}_{AS}\ .\ \alpha\_\texttt{AS}^{\circ}(\{\tau\}) = \{\tau'\}$

Theorem 6. *(Completeness of* $\mathbb{C}_{AS}$*) Let* $\texttt{c}_{AS}$ *be a AsmL class. Then*
$\forall \tau \in \mathcal{T}(\Sigma).\ \tau \in \mathbb{C}_{SC}\,\texttt{c}_{SC} : \exists\ \texttt{AS\_P} \in \langle L_{\texttt{AS\_Pg}}\rangle.\ \exists \rho_0 \in \texttt{AS\_Env} \times \texttt{AS\_Store}.\ \exists\ \texttt{o}_{AS}$ *instance of* $\texttt{c}_{AS}.\ \exists\ \tau' \in \mathcal{T}(\texttt{AS\_Env} \times \texttt{AS\_Store}).\ \tau' \in \mathbb{W}\rho_0 \wedge \alpha\_\texttt{AS}^{\circ}(\{\tau'\}) = \{\tau\}$

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 17. *(SystemC observation function:* α_o^{SC}*)*
Let $\texttt{SC\_Pg} = \langle L_{\texttt{SC\_Mod}}, \texttt{SC\_main}\rangle$ *be a SystemC program, the observation function* $\alpha_o^{SC} \in \mathcal{P}(\texttt{SC\_Env} \times \texttt{SC\_Store}) \rightarrow \mathcal{P}(\mathcal{T}(\texttt{SC\_Env} \times \texttt{SC\_Store}))$ *is*

$$\alpha_o^{SC}\,\texttt{SC\_Pg}(\texttt{R}_0) = lfp^{\subseteq}_{\emptyset}\lambda X.\ \texttt{R}_0 \cup \{\tilde{\rho_0} \rightarrow \ldots \tilde{\rho_n} \mid \forall \tilde{\rho_i} \in (\texttt{SC\_Env} \times \texttt{SC\_Store})\ \exists\ \{\rho_0^i \rightarrow \ldots \rho_m^i\} \in X \wedge \rho_m^i \rightarrow \tilde{\rho_i} \wedge \{\ \texttt{m\_sc}\ in\ \mathbb{M}_{SC} \mid \exists \texttt{o\_sc} \in \mathbb{M}_{SC}.\ \texttt{o\_sc}(\rho_m^i()) \neq \{\epsilon\}\ \} = \emptyset\}$$

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 18. *(AsmL observation function: α_o^{AS})*
Let $\texttt{AS\_Pg} = \langle L_{\texttt{AS\_C}}, \texttt{INIT}\rangle$ *be an AsmL program, the observation function* $\alpha_o^{AS} \in \mathcal{P}(\texttt{AS\_Env} \times \texttt{AS\_Store}) \rightarrow \mathcal{P}(\mathcal{T}(\texttt{AS\_Env} \times \texttt{AS\_Store}))$ *is*

$$\alpha_o^{AS}\texttt{AS\_Pg}(R_0) = lfp\ \overset{\subseteq}{\emptyset}\lambda X.\ (R_0) \cup \{\tilde{\rho_0} \rightarrow \ldots\ \tilde{\rho_n} \mid \forall \tilde{\rho_i} \in (\texttt{SC\_Env} \times \texttt{AS\_Store})\ \exists\ \{\rho_0^i \rightarrow\ \ldots\ \rho_m^i\} \in X\ \wedge\ \rho_m^i \rightarrow \tilde{\rho_i}\ \wedge\ \{\ \texttt{m\_as}\ in\ \mathbb{C}_{AS} \mid \exists \texttt{o\_as} \in \mathbb{C}_{AS}.\ \texttt{o\_as}(\rho_m^i()) \neq \{\epsilon\}\ \} = \emptyset\ \}$$

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 19. *(Equivalence w.r.t. α_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.
$\forall \tilde{\rho} \in \{\tilde{\rho_0} \rightarrow\ \ldots\ \rightarrow \tilde{\rho_n}\} \in \alpha_o^{SC}\texttt{SC\_Pg}(R_0^{SC})$.
$\exists \hat{\rho} \in \{\hat{\rho_0} \rightarrow\ \ldots\ \rightarrow \hat{\rho_n}\} \in \alpha_o^{AS}\texttt{AS\_Pg}(R_0^{AS}) \mid \forall$ vsc $\in$ V_sc. $\exists$ vas $\in$ V_as $\mid$
if vsc $\in$ SC_Sig *then* $\tilde{\rho}$(vsc) = (vl1,vl2) $\wedge$ $\hat{\rho}$(vas) = (vl1,vl2)
if vsc $\in$ AS_DMem *then* $\tilde{\rho}$(vsc) = vl1 $\wedge$ $\hat{\rho}$(vas) =(vl1,vl1)

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 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

Theorem 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

Theorem 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,$\tilde{\rho}$) $\mid \tilde{\rho} \in \alpha_o^{SC}$SC_Pg.
SC_Pg $\vdash$ Prop(V_sc,$\tilde{\rho}$)
: AS_Pg $\vdash$ Prop(V_as,$\hat{\rho}$) $\mid \hat{\rho} \in \alpha_o^{AS}$AS_Pg.
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.*

5 Application: AGP Bus Verification

5.1 Bus Description

AGP (Accelerated Graphics Port) [14] was introduced to meet consumer demand for high-resolution 3D graphics in home computers. New software programs (es-

pecially games) require more and more video bandwidth for fancy textures, high frame rate animations, etc. It has the advantage of allowing large amounts of graphics data to be transferred directly between the computer's main memory and the AGP video card. The AGP bus is designed strictly for video processing and does not have to share available bandwidth with other connected devices. Both AGP bus transactions and PCI bus transactions may be run over the AGP interface. An AGP master (graphics) device may transfer data to the system memory using either AGP transactions or PCI transactions. The corelogic can access the AGP master device only with PCI transactions. Traffic on the AGP interface may consist of a mixture of interleaved AGP and PCI transactions. In addition to the PCI features, AGP includes:
(1) Direct Memory Execute (DME) that gives AGP chips the capability to access the main memory directly for complex operations of texture mapping.
(2) Pipelining and sideband addressing of directly accessing texture maps in system memory.
(3) Multiple requests for data during a bus or memory access.
(4) A dedicated non-shared bandwidth with other devices.

5.2 Model Checking

In order to verify the bus properties, we first used a direct model checking approach by considering a set of properties to verify all the possible transactions scenarios. These cover two main classes: (1) PCI transactions and (2) AGP transactions including both modes DMA and execute. We succeeded to prove the first class of properties with a direct approach while we failed to prove the second set due to state explosion. Therefore, we introduce a proof by induction. Performing the verification of the whole model failed to complete due to a state explosion problem. The main reason for that is the huge size of the read, write and commands queues (each of width 256) present in both the AGP device and the corelogic. By reducing the queues width to three, however, we succeeded to verify all the properties. For more general verification, we defined an induction based approach taking advantage from the abstract data types of AsmL.

We define DRQ: Device Read Queue, DWQ: Device Write Queue, $DReQ$: Device Request Queue, CRQ: Controller Read Queue, CWQ: Controller Write Queue and $CReQ$: Controller Request Queue. The maximum width of the queues is $Q.Wd$. The number of packets in each queue is $XXQ.Np$ (where $XX \in \{DR, DW, DReq, CR, CW, CReq\}$). P is the list of properties under verification.

- Step 1: Verify $P = true$, $\forall$ $DRQ.Np$, $DWQ.Np$, $DReQ.Np$, $CRQ.Np$, $CWQ.Np$, $CReQ.Np \in [0, 1]$.
- Step 2:
  - Hypothesis: Consider $N \in \mathbb{N}$ / $0 < N < Q.Wd$ $\forall x \in \{DRQ.Np, DWQ.Np, DReQ.Np, CRQ.Np, CWQ.Np, CReQ.Np\}$, $x < N : P$ is true.
  - Prove: $\forall x \in \{DRQ.Np, DWQ.Np, DReQ.Np, CRQ.Np, CWQ.Np, CReQ.Np\}$, $x < N + 1 : P$ is true.

5.3 Experimental Results[2]

Model Checking. The CPU time used for the generation of the model checking for queues widths in {1,2,3,6} is given in Table 1. The first three rows are required to ensure the correctness of the initialization conditions. The fourth row, queue width equal to six, is given to illustrate the effect of the numbers of states and transitions increase exponentially as function of the queue size. This clearly illustrates the impossibility of generating the complete FSM for a width of 256. In Table 2.(a) every row corresponds to the proof of a particular queue. Generally, the CPU time, Nodes and number of transitions is close to the case when the queue width is equal to three (see Table 1). Table 2.(b) presents the verification information for the PCI mode which is optional for AGP. A direct proof for this case was possible thanks to the relative simplicity of the PCI, which does not include any queue structure.

Table 1. Validity of Initialization Conditions

Queue width	CPU Time (s)	Number of FSM	
		Nodes	Transitions
1	5.78	34	37
2	30.89	173	193
3	105.20	504	563
6	1758.78	4325	5223

Table 2. Model Checking Results

(a) AGP Mode

Proof for the Queue	CPU Time (s)	Number of FSM	
		Nodes	Trans.
DRQ	341.01	1156	1304
DWQ	345.25	1294	1325
$DReQ$	347.78	1302	1346
CRQ	457.89	1503	1425
CWQ	462.07	1653	1433
$CReQ$	487.01	1859	1481

(b) PCI Mode

Number of		CPU	Number of FSM	
Masters	Slaves	Time (s)	Nodes	Transitions
1	1	2.31	20	25
1	2	2.94	39	53
3	1	26.01	236	341
2	2	26.84	293	449
2	3	101.38	658	1117
3	2	574.18	1881	3153

Assertion Based Verification. We have been able to verify all the AGP bus structure by model checking. However, when the model checking fails, it is possible to use the properties as assertion monitors that can be checked by simulation on the original SystemC model. Using the syntactical transformation defined in [11], we generate the SystemC modules corresponding to the PSL properties. Then, we update the design and integrate the properties as read-only monitors to the global system. We illustrate in Table 3 the simulation statistics

[2] All experiments presented in this section were conducted on a platform consisting of a 2.4 GHz Pentium IV and 512 MB of RAM (PC2700).

Table 3. Simulation Results

Number of		Average Execution
Masters	Slaves	Time per Clock Cycle (10^{-9}s)
1	1	29.321
3	1	32.221
2	2	33.889
2	3	36.568
3	2	38.005
3	3	41.287

of running the new model (combining the original design and the integrated PSL properties) with a random input. The AGP controller can be seen as a slave or a master according to the transaction. The other masters and slaves are just PCI compatible devices. The CPU time confirms the high speed of the SystemC model simulation, which is a direct result from the C++ implementation of the library. Note that the set of assertion monitors including all the properties can be considered as a stand-alone verification IP that can be used to validate other AGP compatible devices either modeled in SystemC or even in Verilog or VHDL.

6 Conclusions

In previous work [10] we introduced a top-down approach similar to the presented in this paper where the verification was integrated as part of the design process and AsmL models were first designed and verified then translated to SystemC. In this paper, we consider a bottom-up approach where starting from an existent SystemC design we generate internally a model in AsmL, an Object-Oriented language used to model systems, and verify the system property at the ASM level. We defined a sound syntactical transformation between SystemC and AsmL to enable model checking at the ASM level. Both the model and its PSL properties were defined in AsmL and checked using a reachability algorithm available in the AsmL tool. We proposed also to translate the same properties used for model checking back to SystemC in order to serve for assertion based verification of the original SystemC design or to serve as a stand-alone verification IP block. We illustrated our approach on the verification of an AGP bus, where we performed a proof by induction to tackle the state explosion problem. Finally, we believe that our approach is an important step towards enabling an efficient formal and semi-formal verification of SystemC. Our future work concerns enhancing the ABV coverage using the FSM generated AsmL models.

References

1. Accellera Organization. Accellera property specification language reference manual, version 1.01. www.accellera.org, 2004.
2. E. Boerger and R. Staerk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer Verlag, 2003.

3. E. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems. In *A Decade of Concurrency - Reflections and Perspectives*, pages 124–175, Berlin, Germany, 1993.
4. 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, USA, 2002.
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 *Proc. International Conference on Software Engineering*, pages 177–187, Toronto, Canada, 2001.
6. W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating finite state machines from abstract state machines. *Software Engineering Notes*, 27(4):112–122, 2002.
7. D. Groβe and R. Drechsler. Checkers for systemc designs. In *Proc. Formal Methods and Models for Codesign*, pages 171–178, San Diego, USA, 2004.
8. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
9. Y. Gurevich, B. Rossman, and W. Schulte. Semantic Essence of AsmL. Technical report, Microsoft Research Tech. Report MSR-TR-2004-27, March 2004.
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*, pages 290–295, Germany, 2005.
11. A. Habibi and S. Tahar. On the Transformation of SystemC to AsmL using Abstract Interpretation. Technical report, ECE, Concordia University, December 2004 (www.ece.concordia.ca/∼habibi/techrp/TR0404/).
12. A. Habibi and S. Tahar. Design for verification of SystemC transaction level models. In *Proc. Design Automation and Test in Europe*, pages 560–565, Germany, 2005.
13. G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.
14. Intel Corp. AGP v3.0 interface specification, 2002.
15. F. Logozzo. *Anhalyse Statique Modulaire de Langages a Objets*. PhD thesis, Ecole Polytechnique, Paris, France, June 2004.
16. Microsoft Corp. AsmL for Microsoft .NET Framework. research.microsoft.com, 2004.
17. Network Processing Forum. *Look-Aside (LA-1) Interface, Implementation Agreement, Revision 1.1*. Kluwer Academic Publishers, April 15, 2004.
18. Open SystemC Initiative. www.systemc.org, 2004.
19. K. Shimizu, D. L. Dill, and A. J. Hu. Monitor-based formal specification of PCI. In *Formal Methods in Computer-Aided Design*, pages 335–353. LNCS 1954, Springer-Verlag, 2000.