Electronic Notes in Theoretical Computer Science 131 (2005) 39–49 

[www.elsevier.com/locate/entcs](http://www.elsevier.com/locate/entcs)

On the Transformation of SystemC to AsmL Using Abstract Interpretation

# Ali Habibi[1](#_bookmark0)

*Electrical and Computer Engineering Concordia University* *Montreal, Canada*

# Sofiene Tahar[2](#_bookmark0)

*Electrical and Computer Engineering Concordia University Montreal, Canada*

Abstract

SystemC is among a group of system level design languages proposed to raise the abstraction level for embedded system design and verification. A straight and sound verification by model checking or theorem proving of SystemC designs is, however, infeasible given the object-oriented nature of this library and the complexity of its simulation environment. We illustrated, in a previous work, the feasibility and success of performing model checking and assertions monitors generation of SystemC using a variant of Abstract State Machines (ASM) languages (AsmL). In this paper, we establish the soundness of our approach by proving the correctness of the transformation from SystemC to AsmL.

*Keywords:* SystemC, Formal Verification, Abstract Interpretation.

# Introduction

SystemC [[11](#_bookmark21)] is an object-oriented system level language for embedded sys- tems 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

1 Email: [habibi@ece.concordia.ca](mailto:habibi@ece.concordia.ca)

2 Email: [tahar@ece.concordia.ca](mailto:tahar@ece.concordia.ca)

1571-0661 © 2005 Elsevier B.V. Open access under [CC BY-NC-ND license.](http://creativecommons.org/licenses/by-nc-nd/3.0/)

doi:10.1016/j.entcs.2005.01.021

SystemC library is composed from 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 HDL (Hardware Description Languages). However, except for small models, the verification of SystemC designs is a serious bottleneck in the system design flow. Direct model checking of Sys- temC designs is not feasible due to the complexity of the SystemC library and its simulator. To solve this problem, we proposed in [[4](#_bookmark11)] to translate SystemC models to an intermediate representation in AsmL [[8](#_bookmark18)] more suitable for formal verification. This approach reduced radically the complexity of the design at the point that we were able to verify a complex PCI architecture using the SMV model checker [[12](#_bookmark22)].

In this paper, we provide a formalization of the SystemC and AsmL se- mantics in fixpoint based on the OO general case given in [[7](#_bookmark16)]. Then, we prove that, for every SystemC program, there exists an AsmL program preserving the same properties, w.r.t. an observation function *α*. The basic concept of this proof of soundness is based on the systematic design of program trans- formation frameworks defined in [[2](#_bookmark12)]. Such a result will enable using a variety of formal tools (for e.g., SMV for model checking [[12](#_bookmark22)]) or to use AsmL tool (Asmlt) to generate a finite state machine of the design.

Related work to ours concerns in particular, defining the formal semantics of SystemC and AsmL. For instance, several approaches have been used to write the SystemC semantics (e.g., using ASM is [[9](#_bookmark19)]). Denotational semantics

[[10](#_bookmark20)] is found to be most effective since objects can be expressed as fixpoints on suitable domains. Salem in [[13](#_bookmark23)] proposed a denotational semantics for SystemC. However, the proposal in [[13](#_bookmark23)] 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 define, in this paper, our own SystemC denotational semantics.

Regarding, the program transformation, the work of Patrick and Radhia Cousot in [[2](#_bookmark12)] is the essence for any program transformation using abstract interpretation. The tactical choice of using semantics to link the subject pro- gram 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. A projection of that generic approach, described in Section 3.9 of

[[2](#_bookmark12)] 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 need to define the syntax, semantics and observation functions for both AsmL and SystemC.

The rest of this paper is organized as follows: Section [2](#_bookmark1) and Section [3](#_bookmark6)

present, respectively, the SystemC and AsmL semantics in fixpoint. Section [4](#_bookmark8) contains the proof of the existence and soundness of the SystemC to AsmL transformation. Finally, Section [5](#_bookmark10) concludes the paper.

# SystemC Fixpoint Semantics

* 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 2.1 (SystemC Module: SC Module)

A SystemC Module is a set ⟨DMem, Ports, Chan, Mth, SC Ctr⟩, 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 2.2 (SystemC Port: SC Port)

A SystemC Port is a set ⟨IF, N, SC In, SC Out, SC InOut⟩, 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 functions).

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 2.3 (SystemC Constructor: SC Ctr)

A SystemC Constructor is a set ⟨Name, Init, SC Pr, SC SSt⟩, 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 2.4 (SystemC Process: SC Pr)

A SystemC process is a set ⟨PMth, PTh, PCTh⟩, where PMth is a method process (defined as a set ⟨Mth, SC SL⟩ including the method and its sensitivity list), PTh is a thread process (accepts a wait statement in comparison to the method process), PTh is a clocked thread process (sensitive to the clock event).

Definition 2.5 (SystemC Program: SC Pg)

A SystemC program is a set ⟨LSC Mod, SC main⟩, where LSC 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.

* 1. *Fixpoint Semantics*

In this section, we define the semantics of the whole SystemC program, W

〚SC Pg〛, and the SystemC module, M*SC*〚m sc〛. Then, present the proofs (or proof sketches) of the soundness and completeness of M*SC*〚m sc〛.

Definition 2.6 (Delta Delay: *δd*)

The SystemC simulator considers two phases *evaluate* and *update*. The sepa- ration between these two phases is called *delta delay*.

Definition 2.7 (SystemC Environment: SC Env)

The SystemC environment is the summation of the default C++ environment (Env) as defined in [[7](#_bookmark16)] and the signal environment (Sig Store) specific to SystemC: SC Env = 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

⊆ N is a set of addresses.

Definition 2.8 (SystemC Store: SC Store)

The SystemC store is the summation of the default C++ store (Store) as de- fined in [[7](#_bookmark16)] 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 R0 ∈ P(SC Env×SC Store) be a set of initial states, pcin be the en- try point of the main function sc main and →⊆: (SC Env × SC Store) × (SC Env×SC Store) be a transition relation.

Definition 2.9 (Whole SystemC Program Semantics: W 〚SC Pg〛)

Let SC Pg = ⟨LSC Mod, SC main⟩ be a SystemC program. Then, the semantics of SC Pg, W 〚SC Pg 〛∈ P(SC Env×SC Store) → P(T (SC Env× SC Store)) is

W〚SC Pg〛(R0) = lfp ⊆*λX*. (R0) ∪ {*ρ*0 → *... ρn* → *ρn*+1| *ρn*+1 ∈ (SC Env×

∅

SC Store) ∧ {*ρ*0 → *... ρn*} ∈ *X* ∧ *ρn* → *ρn*+1}

Both definitions of the semantics of process declaration (P*R* 〚SC Pr〛) and SystemC module constructor (P*Ctr* 〚SC Ctr〛) are given in [[6](#_bookmark17)]. In contrast to the semantics definition of an OO object in [[7](#_bookmark16)], 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 (O*SC*〚o sc〛) through the definition of a transition

function nextsc(*σ*)=next(*σ*) nextsig(*σ*), including both parts C++ re- lated and SystemC specific functions, can be found in [[6](#_bookmark17)].

Theorem 2.10 *Let*

*F = λT. S*

⟨*v, s*⟩ ∪ {*σ*

*l*0 *l*n−1 *l*n

*... σ σ* |

'

*sc* *0*

0 → → *n* →

*l*0 *l*n−1

{*σ ... σ*

∈ *T, next*

*(σ )* e ⟨*σ*'*, l*'⟩}

0 → → *n*

*sc n*

*Then* O*SC*〚*o sc*〛*)*(*vsc, ssc*) *=* ∪*ω Fscn*(∅)

*n*=0

Proof. The proof is immediate from the fixpoint theorem in [[1](#_bookmark13)].

Definition 2.11 (SystemC Module Semantics: M*SC*〚m sc〛))

Let m sc = ⟨DMem, Ports, Chan, Mth, SC Ctr⟩ be a SystemC module, then its semantics M*SC*〚m sc〛) ∈ P(T (Σ)) is:

M*SC*〚m sc〛= {O*SC*〚o sc〛(*vsc, ssc*) | o sc is an instance of m sc, v sc ∈ D in, s sc ∈ SC Store}

Theorem 2.12 *(SystemC Module semantics in ﬁxpoint) Let*

*l*0 *l*n−1

*l*' '

*Gsc*⟨*S*⟩*= λT.* {*S0*⟨*v, s*⟩ | ⟨*v, s*⟩∈ *S* } ∪ {*σ*0 → *...* → *σn* → *σ* |

*l*0 *l*n−1

{*σ ... σ*

∈ *T, next*

*(σ )* e ⟨*σ*'*, l*'⟩}

0 → → *n*

*sc n*

*Then* M*SC*〚*m sc*〛(*vsc, ssc*) *= lfp* ⊆ *Gsc*⟨ *Din*×*Store*⟩

∅

Proof. Although the SystemC model presents some additional functionalities on top of C++, the proof of this theorem is similar to the proof of Theorem

3.2 in [[7](#_bookmark16)]. For instance, considering the definition of M*SC* and applying in order Definition of a SystemC module object in [[6](#_bookmark17)], Theorem [2.10](#_bookmark2) and the fixpoint theorem in [[1](#_bookmark13)], the proof is straightforward.

The last step in the SystemC fixpoint semantics is to relate the module semantics to the whole SystemC program semantics. Hence, we consider up- dated version of the function *abstract* ( *α*◦) as defined in [[7](#_bookmark16)]. The new function is upgraded to support the SystemC simulation semantics, environment and store. The complete definitions of *α* SC◦ can be found in [[6](#_bookmark17)].

Theorem 2.13 *(Soundness of* M*SC*〚*m sc*〛*) Let MSC be a whole SystemC pro- gram and let mSC* ∈ *MSC. Then*

∀ *R*0 ∈ *SC Env*× *SC Store.* ∀ *τ* ∈ T *(SC Env*× *SC Store).*

*τ* ∈ W〚*SC Pg*〛(*R*0) : ∃*τ* ' ∈ M*SC*〚*mSC* 〛*. α SC*◦({*τ* })= {*τ* '}

Proof. (Sketch) We have to consider both cases when *τ* contains an object o*SC*, instantiation of m*SC*, and when it does not include any o*SC*. For the second situation, the proof of the theorem is trivial considering that *τ* will be an empty trace. In the first case, the trace is not empty (let it be *τ* ''). Since

SystemC modules are initialized in the main program sc main before the simulation starts, there exist an initial environment, store and set of variables that define the initial trace *σ*0 ∈ *τ* ''. The rest of the traces in *τ* '' are interaction states of o*SC* because they are obtained by applying *α* SC◦ on *τ* . Therefore, *τ* '' ∈ M*SC*〚m*SC*〛.

Theorem 2.14 *(Completeness of* M*SC*〚〛*) Let mSC be a SystemC module. Then*

∀*τ* ∈ T (Σ)*. τ* ∈ M*SC*〚*mSC* 〛: ∃ *SC P* ∈ ⟨*LSC Pg*⟩*.* ∃*ρ*0 ∈ *SC Env*×

*SC Store.* ∃ *oSC instance of mSC. exists τ* ' ∈ T *(SC Env*× *SC Store). τ* ' ∈ W〚*ρ*0〛∧ *α SC*◦({*τ* '})= {*τ* }

Proof. (Sketch) A SystemC program satisfying the previous theorem can be constructed by creating and instance of m*SC* in the sc main function, the initial state corresponds to the state when the module’s constructor, SC Ctr, was executed. An execution of a method of m*SC* corresponds to executing a method thread (setting of the events in its sensitivity list to *Active*) and a change of a port corresponds to updating its internal signal by the new values. Hence, it is always possible to construct both SC P and *ρ*0. For instance, there exist many other possible constructions involving SystemC threads, clocked threads, etc.

# AsmL Fixpoint Semantics

AsmL [[8](#_bookmark18)] is one of the very latest languages developed for ASM [[3](#_bookmark14)]. It supports object-oriented modeling at higher level of abstraction in comparison to C++ or Java. We are going to restrict the AsmL semantics presented in this paper to the subset used in the program transformation.

* 1. *Syntactical Domains*

Definition 3.1 (AsmL Class: AS C)

An AsmL class is a set ⟨AS DMem, AS Mth, AS Ctr⟩, 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 corre- sponds to the methods pre-conditions (Boolean proposition verified before the execution of the method).

Definition 3.2 (AsmL Method: AS Mth)

An AsmL method is a set ⟨AS M, AS Pre, AS Pos, AS Cst⟩, where AS M is a the

core of the method, 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 3.3 (AsmL Program: AS Pg)

An AsmL Program is a set ⟨LAS C, INIT⟩, where LAS C is a set of AsmL classes and INIT is the main function in the program.

* 1. *Fixpoint Semantics*

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

Definition 3.4 (AsmL Environment: AS Env)

The AsmL Environment is a modified OO environment AS Env = [Var → Addr,Addr], where Var is a set of variables and Addr ⊆ N is as set of addresses (two addresses store the current and new values of v ∈ Var).

Definition 3.5 (AsmL Store: AS Store)

The AsmL store is AS Store = [(Addr, Addr) → (Val,Val)], where Val is a set of values such that AS Env ⊆ Val.

The whole AsmL program semantics (W*AS* 〚AS Pg〛), method semantics (M*AS* 〚. 〛) and object semantics (O*AS*〚o AS〛) through the definition of a transition function nextas(*σ*) can be found in [[5](#_bookmark15)]. The AsmL class constructor is a can be defined according to the Definition 3.8 in [[7](#_bookmark16)].

Definition 3.6 (AsmL Class Semantics: C*AS*〚c as〛)

Let c as = ⟨as dmem, as mth, as ctr⟩ be an AsmL class, then its semantics

C*AS*〚c as〛) ∈ P(T (Σ)) is:

C*as*〚c as〛= {O*AS*〚o as〛(*vas, sas*) | o as is an instance of c as, v as ∈ D in,

s as ∈ SC Store}

Theorem 3.7 *(AsmL Class semantics in ﬁxpoint) Let*

*l*0 *l*n−1

*l*' '

*Has*⟨*S*⟩*= λT.* {*S0*⟨*v, s*⟩ | ⟨*v, s*⟩∈ *S* } ∪ {*σ*0 → *...* → *σn* → *σ* |

*l*0 *l*n−1

{*σ ... σ*

∈ *T, next*

*(σ )* e ⟨*σ*'*, l*'⟩}

0 → → *n*

*as n*

*Then* C*AS*〚*c as*〛(*vas, sas*) *= lfp* ⊆ *Has*⟨ *Din*×*Store*⟩

∅

Proof. Similar to the proof of Theorem [2.12](#_bookmark3)

The function *α* AS◦ is an updated version of the function *abstract* (*α*◦) defined in [[7](#_bookmark16)]. The complete definition of *α* AS◦ is given in [[5](#_bookmark15)].

Theorem 3.8 *(Soundness of* C*AS*〚*c as*〛*) Let PAS be a whole SystemC pro- gram and let cSC* ∈ *CSC. Then*

∀ *R*0 ∈ *AS Env*× *AS Store.* ∀ *τ* ∈ T *(AS Env*× *AS Store).*

*τ* ∈ W〚*AS Pg*〛(*R*0) : ∃*τ* ' ∈ C*AS*〚*cAS* 〛*. α AS*◦({*τ* })= {*τ* '}

Proof. Similar to Theorem [2.13](#_bookmark4).

Theorem 3.9 *(Completeness of* C*AS*〚〛*) Let cAS be a SystemC module. Then*

∀*τ* ∈ T (Σ)*. τ* ∈ C*SC*〚*cSC* 〛: ∃ *AS P* ∈ ⟨*LAS Pg*⟩*.* ∃*ρ*0 ∈ *AS Env*×

*AS Store.* ∃ *oAS instance of cAS. exists τ* ' ∈ T *(AS Env*× *AS Store). τ* ' ∈ W〚*ρ*0〛∧ *α AS*◦({*τ* '})= {*τ* }

Proof. Similar construction approach to what we proposed in Theorem [2.14](#_bookmark5) except that instead of considering the SystemC method thread and sensitivity list, we consider here AsmL methods and their pre-conditions. Here also, there exist many possible constructions involving AsmL post-conditions, for example.

# Program Transformation

The equivalence in behavior, with respect to the observation *αo*, 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 program by a method (method core, pre-condition) in the AsmL program to ensure having set of variables in both programs updated in the same time with the same values.

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 4.1 (SystemC observation function: *αSC*)

*o*

Let SC Pg= ⟨LSC Mod, SC main⟩ be a SystemC program, the observation func- tion *αSC* ∈ P(SC Env×SC Store) → P(T (SC Env× SC Store)) is

*o*

*αSC*〚SC Pg〛(R0) =

*o*

lfp ⊆*λX*.(R0) ∪ {*ρ*˜0 → *... ρ*˜*n*| ∀*ρ*˜*i* ∈ (SC Env× SC Store) ∃ {*ρi* → *... ρi* }

∅ 0 *m*

∈ *X* ∧ *ρi* → *ρ*˜*i* ∧ { m sc in M*SC* | ∃o sc ∈ M*SC.* o sc(*ρi* ()) /= {*ϵ*} } = ∅}

*m*

*m*

In the previous definition, *αSC* is only tracing the initial states of a simu- lation cycle. For instance, the third condition confirms that in the last simu- lation cycle there was no single process ready to run. Similarly, we define an observation function *αAS* for an AsmL program.

*o*

*o*

Definition 4.2 (AsmL observation function: *αAS*)

*o*

*AS*

Let AS Pg= ⟨LAS C, INIT⟩ be an AsmL program, the observation function *α*

*o*

∈ P(AS Env×AS Store) → P(T (AS Env× AS Store)) is

*αAS*〚AS Pg〛(R0) =

*o*

lfp ⊆*λX*.(R0) ∪ {*ρ*˜0 → *... ρ*˜*n*| ∀*ρ*˜*i* ∈ (SC Env×AS Store) ∃ {*ρi* → *... ρi* }

∅ 0 *m*

∈ *X* ∧ *ρi* → *ρ*˜*i* ∧ { m as in C*AS* | ∃o as ∈ C*AS.* o as(*ρi* ()) /= {*ϵ*} } = ∅ }

*m*

*m*

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

Definition 4.3 (Equivalence w.r.t. *αo*: ≡*α*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 ≡*α*o prog as if

∀R*SC* set of initial states of SC Pg. ∀R*AS* set of initial states of AS Pg.

0 0

∀*ρ*˜ ∈ {*ρ*˜0 → *...* → *ρ*˜*n*} ∈ *αSC*〚SC Pg〛(R*SC*).

*o* 0

∃*ρ*ˆ ∈ {*ρ*ˆ0 → *...* → *ρ*ˆ*n*} ∈ *αAS*〚AS Pg〛(R*AS*) |

*o* 0

∀ vsc ∈ V sc. ∃ vas ∈ V as such that

if vsc ∈ SC Sig then (*ρ*˜(vsc) = (vl1,vl2)) ∧ (*ρ*ˆ(vas) = (vl1,vl2)) if vsc ∈ AS DMem then (*ρ*˜(vsc) = vl1) ∧ (*ρ*ˆ(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 4.4 *(Existence of transformed AsmL program w.r.t. αSC) Let SC Pg*

*o*

*be a whole SystemC program, SC Din a set of inputs and SC Dout a set of out-*

*puts. Then*

∃ *AS Pg, an AsmL program, such that SC Pg* ≡*α*o *AS Pg*

Proof. (Sketch) The proof is done by constructing the AsmL program. For instance, for every SystemC module we affect an AsmL class having the same data members and methods. We set the pre-conditions, AS Ctr, for the AsmL methods as a conjunction of the state of the events present in the sensitivity list, SC SL, of the SystemC program processes. The tricky point in the con- struction is when to make the updates in the AsmL program. We have two possibilities: (1) C++ variables update: whenever a C++ variable is involved in an instruction, a partial update can be applied using the notion of *binders* in AsmL; and (2) SystemC signals: all signals are updated when all methods pre-conditions are false. Once the set of AsmL classes defined, Theorem [3.9](#_bookmark7) ensures the existent of the AsmL program.

Theorem 4.5 *(Soundness of the transformation) Let SC Pg be a whole Sys- temC program and let AS Pg be a whole AsmL program. Then*

*SC Pg* ≡*α*o *AS Pg* : ∀ *Prop(V sc,ρ*˜*)* | *ρ*˜ ∈ *αSC*〚*SC Pg*〛*.*

*o*

*SC Pg* ▶ *Prop(V sc,ρ*˜*)* : *AS Pg* ▶ *Prop(V as,ρ*ˆ*)* | *ρ*ˆ ∈ *αAS*〚*AS Pg*〛*.*

*o*

*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. The proof is straightforward from the construction of equivalence re- lation ≡*α*o in Definition [4.3](#_bookmark9).

# Conclusion

In this paper, we presented the fixpoint semantics of the SystemC library including, in particular, the semantics of a SystemC Module that we proved to be soundn 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

[[4](#_bookmark11)] using model checking and theorem proving approaches used with AsmL and ASM languages in general.

# References

1. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, TX, USA, 1979.
2. P. Cousot and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *Symposium on Principles of Programming Languages*, pages 178– 190, Portland, Oregon, January 2002.
3. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In *Speciﬁcation and Validation Methods*, pages 9–36. Oxford University Press, 1995.
4. A. Habibi and S. Tahar. Design for verification of SystemC transaction level models. In *Design Automation and Test in Europe*, Munich, Germany.
5. A. Habibi and S. Tahar. AsmL fixpoint semantics. Technical report, Department of ECE, Concordia University, December 2004.
6. A. Habibi and S. Tahar. SystemC fixpoint semantics. Technical report, Department of ECE, Concordia University, December 2004.
7. F. Logozzo. *Analyse Statique Modulaire de Langages `a Objets*. PhD thesis, Ecole Polytechnique, Paris, France, June 2004.
8. Microsoft Corp. AsmL for Microsoft .NET framework, 2004.
9. W. Mu¨ller, J. Ruf, and W. Rosenstiel. *SystemC Methodologies and Applications*. Kluwer Academic Pub., 2003.
10. P. D. Mosses. *Denotational semantics*, volume B of *Handbook of Theoretical Computer Science*, chapter 11, pages 575–631. Elsevier Science B.V., 1990.
11. Open SystemC Initiative. Website: [http://www.systemc.org,](http://www.systemc.org/) 2004.
12. K. Oumalou, A. Habibi, and S. Tahar. Design for verification of a PCI bus in SystemC. In

*Symposium on System-on-Chip*, Finland, November 2004.

1. A. Salem. Formal semantics of synchronous SystemC. In *Design, Automation and Test in Europe Conference*, pages 376–381, Munich, Germany, 2003.