AsmL Semantics in Fixpoint

Ali Habibi and Sofiène Tahar

Department of Electrical and Computer Engineering
Concordia University
1455 de Maisonneuve, West,
Montreal, Quebec H3G 1M8
{habibi,tahar}@ece.concordia.ca

Abstract. AsmL is a novel executable specification language based on the theory of Abstract State Machines (ASMs). It represents one of the most powerful practical engines to write and execute ASMs. In this paper, we present a proven complete small-step trace-based operational semantics of the main parts of AsmL. Such a semantics provides precise and non ambiguous definitions of AsmL. It is very useful to guarantee a unique implementation of the language and interpretation of its behavior. Furthermore, they can be used in conducting formal proofs for sound abstractions or even to construct syntactical transformers to other languages.

1 Introduction

AsmL (the Abstract State Machine Language) [5] is a novel executable specification language based on the theory of Abstract State Machines (ASMs). It is fully object-oriented (OO) and has a strong mathematical component. In particular, sets, sequences, maps and tuples are available as well as set comprehension, sequence comprehension and map comprehension. ASMs steps are transactions, and in that sense AsmL programming is transaction based. AsmL is fully integrated into the .NET framework and Microsoft development tools providing inter-operability with many languages and tools.

Although the language features of AsmL were chosen to give the user a familiar programming paradigm (supporting classes and interfaces in the same way as C# or Java do), the crucial features of AsmL, intrinsic to ASMs, are massive synchronous parallelism and finite choice. These features give rise to a cleaner programming style than is possible with standard imperative programming languages. Synchronous parallelism and inherently AsmL provide a clean separation between the generation of new values and the committal of those values into the persistent state.

An operational semantics of AsmL, in ASMs, was defined in [5] for a subset of AsmL called AsmL-S (a core of AsmL). However, the use of ASM as a concrete semantics has two main drawbacks. First, the ASM notation has the tendency to hide low-level details, by making wide use of macros. While this may be an advantage to the casual reader, it is a drawback for the design of precise yet sound static analyses. Second, the program computation is hidden in the ASM transition relation, and the fixpoint computation is not explicitly stated. As a consequence, this formalism is inadequate to express, for example, program-wide invariant properties.

Denotational semantics is well-suited for modeling object-oriented languages, where both object's self application and inheritance can be smartly expressed as fixpoints on suitable domains [8]. Moreover, it is straightforward to consider a domain composed by an environment (a map from variables to addresses) and a store (a map from addresses to values). Hence, object aliasing can be naturally expressed. It was shown in [2] that generally denotational semantics is an abstraction of a trace-based operational semantics in the sense that it abstracts away the history of computations, by considering input-output functions. As a consequence, in this paper, we provide a formalization of the AsmL small-step operational semantics. For instance, we will first enumerate the syntactical domains. Then, we will provide the semantics of whole program and the semantics of AsmL classes. Finally, we will provide the proofs for the completeness and soundness of the whole semantics. Our approach updates on the generic semantics provided by Logozzo in [9] by: (1) modifying the syntactical domains to support AsmL specific domains; (2) upgrading the default environment and store to include the values of the variables in the update and evaluate phases; and (3) re-establishing the soundness and completeness proofs considering the updates and modifications of points (1) and (2).

The rest of the paper is organized as follows: Section 2 describes the main AsmL syntactical and semantical domains. Section 3 presents the AsmL semantics in fixpoint. Section 4 shows the proofs of correctness and completeness of the proposed AsmL semantics. Section 5 gives an application of the proposed semantics. Finally, Section 6 concludes the paper.

2 Syntactical and Semantical Domains

2.1 Syntactical Domains

We will present the basic syntactical domains that are required for the semantics section. These include: classes, methods, constraints and programs.

A class is a description for a set of objects. The programmer specifies the class constructor, methods and fields (data members). Hence, we define an AsmL class as:

Definition 1. *(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 class data members,* `AS_Mth` *a set of methods (functions) definition and* `AS_Ctr` *is the class constructor.*

One of the important AsmL features corresponds to the methods pre-conditions (Boolean proposition verified before the execution of the method). These conditions distinguish AsmL methods from default OO methods (e.g., Java methods).

Definition 2. *(AsmL Method:* `AS_Mth`*)*
An AsmL method is a set $\langle$`AS_M`*,* `AS_Pre`*,* `AS_Pos`*,* `AS_Cst`$\rangle$*, 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. *(AsmL Method Precondition:* `AS_Pre`*)*
An AsmL method pre-condition is a set $\langle$`AS_B`$\rangle$*, where* `AS_B` *is a Boolean proposition.*

An AsmL program can be seen as a set of classes. In particular, it can be made up by a main class and a library of classes according to:

Definition 4. *(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.*

2.2 Semantical Domains

The first step in the specification of the AsmL semantics is the definition of the semantical domains. For instance, we need to define a unique identity and environment for every object. The general way to fulfill such a requirement is to consider a domain of <environment, store> pairs; where, an environment maps a variable name to a memory address and a store maps a memory address to a memory element.

AsmL is deferent from general OO languages in the sense that it 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 5. *(Total Update:* `Step`*)*
A total update is performed using the `Step` *instruction and affects all the programs variables.*

Considering the update notion of AsmL, we formalize its environment as:

Definition 6. *(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.*

For every variable correspond two addresses storing its current and the new values.

Definition 7. *(AsmL Store: `AS_Store`)*
The AsmL store is $\mathtt{AS\_Store} = [(\mathtt{Addr},\ \mathtt{Addr}) \rightarrow (\mathtt{Val},\mathtt{Val})]$*, where* `Val` *is a set of values such that* $\mathtt{AS\_Env} \subseteq \mathtt{Val}$.

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

3 Fixpoint Semantics

3.1 Whole AsmL Program Semantics

The whole AsmL program semantics can be defined as the traces of the executions of the program starting from a set of initial states $\mathtt{R}_0$. It can be expressed in fixpoint semantics as follows:

Definition 8. *(Whole AsmL Program Semantics:* $\mathbb{W}_{AS}$ $[\![\mathtt{AS\_Pg}]\!]$*)*
Let $\mathtt{AS\_Pg} = \langle L_{\mathtt{AS\_C}}, \mathtt{Main}\rangle$ *be an AsmL program. Then, the semantics of* `AS_Pg`*,* $\mathbb{W}_{AS}$ $[\![\mathtt{AS\_Pg}]\!] \in \mathcal{P}(\mathtt{AS\_Env}\times\mathtt{AS\_Store}) \rightarrow \mathcal{P}(\mathcal{T}(\mathtt{AS\_Env}\times\ \mathtt{AS\_Store}))$ *is:*

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

The trace semantics of an AsmL method states that at the maximal trace: (1) all *pre*−, *post*− and *pct*− are evaluated to *true*; (2) the program counter refers to the end point of the method; and (3) the height of the stack is the same at the entry of the method.

Definition 9. *(Method Semantics:* $\mathbb{M}_{AS}$ $[\![.\ ]\!]$*))*
Let $\mathtt{AS\_Mth} = \langle \mathtt{AS\_M},\ \mathtt{AS\_Pre},\ \mathtt{AS\_Pos},\ \mathtt{AS\_Cst}\rangle$ *be an AsmL method. Then, the semantics of* `AS_Mth`*,* $\mathbb{M}_{AS}$ $[\![\mathtt{AS\_m}\ ]\!]) \in \mathcal{P}(\mathtt{AS\_Env}\times\mathtt{AS\_Store}) \rightarrow \mathcal{P}(\mathcal{T}(\mathtt{AS\_Env}\times \mathtt{AS\_Store}))$ *is*

$$\begin{aligned}&\mathbb{M}_{AS}\ [\![\mathtt{AS\_m}]\!](R_0, M, Pre, Pos, Cst) = \\ &\quad lfp\,^{\subseteq}_{\emptyset}\ \lambda X,\ m,\ spre,\ spos,\ scst.\ (R_0) \cup \{\ \rho_0 \rightarrow\ \ldots\ \rho_n \rightarrow \rho_{n+1}|\ \rho_{n+1} \in (\mathtt{AS\_Env}\times\ \mathtt{AS\_Store}) \wedge \\ &\qquad \{\rho_0 \rightarrow\ \ldots\ \rho_n\} \in X \wedge \\ &\qquad \rho_n \rightarrow \rho_{n+1} \wedge \\ &\qquad \rho_{n+1}(X) = (m, spre,\ spos, scst) \\ &\qquad \wedge\ spre = spos = scst = true\}\end{aligned}$$

3.2 AsmL Class Semantics

The semantics of an AsmL class can be seen as the set of all the semantics of its instances. The semantics of an object is the set of traces that correspond to the evolution of the object internal state. The AsmL class constructor is a default OO constructor. It can be defined according to the Definition 3.8 in [9].

In a general OO context, such as Java, an object can be defined as a set of states including a first (initial) state representing the object just after its creation and a set of states resulting from the interaction of the object with its context [9]. In this case, the interaction can happen in two ways: (1) the context invokes an object's method, or (2) the context modifies a memory location reachable from the object's environment. In [9], this interaction was very well defined using two functions $\mathtt{next_d}$, for direct interactions, and $\mathtt{next_{ind}}$ for indirect interactions and the object semantics, $\mathbb{O}\,[\![\mathtt{o}]\!]$, was defined as:

$$\mathbb{O}[\![\mathtt{o}]\!](\mathtt{v,s}) = \mathrm{lfp}\,^{\subseteq}_{\emptyset}\ \lambda T.\ \mathtt{S_0}\langle v, s\rangle \cup \{\sigma_0 \overset{l_0}{\rightarrow} \dots \overset{l_{n-1}}{\rightarrow} \sigma_n \overset{l_n}{\rightarrow} \sigma' \,|\, \{\sigma_0 \overset{l_0}{\rightarrow} \dots \overset{l_{n-1}}{\rightarrow} \sigma_n \in T,\ \mathtt{next}(\sigma_n)\ \ni\ \langle \sigma', l'\rangle\}$$

where $\mathrm{next}(\sigma) = \mathtt{next_d}(\sigma) \cup \mathtt{next_{ind}}(\sigma)$, l is a transition label and $\mathtt{S_0}\langle v, s\rangle$ is a set of initial states.

In addition to the semantics definition of an OO object in [9], an AsmL method can be activated by an update instruction. This interaction is a hybrid direct/indirect interaction because concerned methods will be invoked according to the state of the program events (that maybe external to the object). In following, we will define the interaction states, then, we will provide the complete definition for the direct, indirect and AsnL specific interaction functions.

Definition 10. *(Interaction States)*
The set of interaction states is $\sum = \mathtt{AS\_Env} \times \mathtt{AS\_Store} \times \mathtt{D_{out}} \times \mathcal{P}(\mathtt{Addr})$

After the creation of the object, the reached states represent the initial states defined as follows:

Definition 11. *(Initial States* $\mathtt{S_0}\langle v_{as}, s_{as}\rangle$*)*
Let $v_{as} \in \mathtt{D_{in}}$ *be an AsmL object input value,* $s_{as} \in \mathtt{AS\_Store}$ *a store at object creation time and* $\mathtt{AS\_Obj}$ *an AsmL object. The set of initial states of* $\mathtt{AS\_Obj}$ *is:*

$$\mathtt{S_0}\langle v_{as}, s_{as}\rangle = \{\langle e'_{as}, s'_{as}, \phi, \emptyset\rangle \mid \mathbb{P}_{Ctr}[\![\mathtt{AS\_Ctr}]\!](\mathtt{L}_M) \ni \langle e'_{as}, s'_{as}\rangle\}$$

where: $\mathtt{L}_M = \{m_1, \dots, m_i, \dots, m_n\}$ *is a list of the the methods.*

In Definition 11 ϕ is a void value ($\in \mathtt{D_{out}}$) meaning that the constructor does not return any value and therefore does not expose any address to the context.

For every state we tag to transitions to its successors by a label. This latter can consist of the name of the invoked method (and its input values) or a term to denote an action from the context according to the following definition:

Definition 12. *(Transition Labels:* $\mathtt{Label\_AS}$*)*
The set of transition labels is $\mathtt{Label\_AS} = (\mathtt{Mth} \times \mathtt{D_{in}}) \cup \{k\}$.

In Definition 12 we distinguish two types of interactions corresponding respectively to: (1) invoking a method (direct interaction); and (2) modifying the memory location that is reachable from the the object environment (indirect interaction). The transition function `next_AS` is made up of two functions: $\texttt{next\_AS}_{\texttt{dir}}$ and $\texttt{next\_AS}_{\texttt{ind}}$.

Definition 13. *(Direct interactions: $\mathit{next\_AS}_{\mathit{dir}}$)*
Let $\langle e_{as}, s_{as}, v_{as}, \mathit{Esc}\rangle \in \sum$ *an interaction state. Then, the direct interaction function* $\mathit{next\_AS}_{\mathit{dir}} \in [\sum \rightarrow \mathcal{P}(\sum \times \mathit{Label\_AS})]$ *is defined as:*

$$\begin{aligned} \mathit{next\_AS}_{\mathit{dir}}(\langle e_{as}, s_{as}, v_{as}, \mathit{Esc}\rangle) = \{ & \langle\langle e'_{as}, s'_{as}, v'_{as}, \mathit{Esc'}\rangle, \langle mth, v_{in}\rangle\rangle \\ & \mid mth \in \mathit{Mth},\ v_{in} \in \mathit{Din}, \\ & \mathbb{C}[\![\mathit{mth}]\!](v_{in}, e_{as}, s_{as}) \ni (v'_{as}, e'_{as}, s'_{as}), \\ & \mathit{Esc'} = \mathit{Esc} \cup \text{reachable}(v'_{as}, e'_{as})\}. \end{aligned}$$

where $\mathbb{C}[\![\mathit{mth}]\!]$ *is the semantics of generic OO method as defined in [9].*

The function *reachable* is an extension of the helper function defined in [9]. For instance, given an address v_{as} and a store s_{as}, *reachable* determines all the addresses that are reachable from v_{as}. In the AsmL context, this function acts only on the data members of the class according to the following recursive definition:

Definition 14. *(The function* reachable*)*
The function reachable $\in [D_{out} \times \mathit{AS\_Store}] \rightarrow \mathcal{P}(\mathit{Addr})$ *is defined as follows:*

$$\begin{aligned} & \text{reachable}(v_{as}, s_{as}) = \\ & \quad \textit{if } v_{as} \in \mathit{Addr} \textit{ then} \\ & \qquad \{\mathit{Addr}\} \cup \{\text{reachable}(e'_{as}(d_{mem}), s'_{as}) \mid \exists\ \mathit{as\_class} = \\ & \qquad \langle \mathit{AS\_DMem}, \mathit{AS\_Mth}, \mathit{AS\_Ctr}\rangle,\ d_{mem} \in \mathit{AS\_DMem},\ s_{as}(v_{as}) \\ & \qquad \textit{is an instance of } \mathit{as\_class},\ s_{as}(s_{as}(v_{as})) = e'_{as}\} \\ & \quad \textit{else } \emptyset. \end{aligned}$$

The second possible interaction corresponds to indirect interaction, which may happen when an address escapes from an object. In that case, the context can modify the content of this address with any value. The function $\texttt{next\_AS}_{\texttt{ind}}$ defines this type of interaction:

Definition 15. *(Indirect interactions: $\mathit{next\_AS}_{\mathit{ind}}$)*
Let $\langle e_{as}, s_{as}, v_{as}, \mathit{Esc}\rangle \in \sum$ *an interaction state. Then, the indirect interaction function* $\mathit{next\_AS}_{\mathit{ind}} \in [\sum \rightarrow \mathcal{P}(\sum \times \mathit{Label\_AS})]$ *is defined as:*

$$\begin{aligned} \mathit{next\_AS}_{\mathit{ind}}(\langle e_{as}, s_{as}, v_{as}, \mathit{Esc}\rangle) = \{ & \langle\langle e_{as}, s'_{as}, \phi, \mathit{Esc}\rangle, k\rangle \mid \\ & \exists\ \alpha \in \mathit{Esc}.\ s'_{as} \in \text{update\_as}(\alpha, s'_{as})\} \end{aligned}$$

The *update_as* function is an extension of the *update* function defined in [9] in the sense that it considers AsmL updates in addition to variables. It is defined as follows:

Definition 16. *(The function* update_as*)*
The function update_as $\in$ [*Addr* $\times$ *AS_Store* $\rightarrow \mathcal{P}($*AS_Store*$)$] *is defined as follows:*

$$\text{update\_as}(\alpha, s_{as}) = \{s'_{as} \mid \exists\, v \in \texttt{Val}.\ s'_{as} = s_{as}[\alpha \mapsto v]\ \}.$$

update_as returns all the possible stores, where $s_{as}(\alpha)$ takes all the possible values in the values domain `Val`.

Using the definitions of $\texttt{next\_AS}_{\texttt{dir}}$ and $\texttt{next\_AS}_{\texttt{ind}}$, we define the global transition function `next_AS` as:

Definition 17. *(Transition function: next_AS)*
Let $\texttt{st} = \langle e_{as}, s_{as}, v_{as}, \texttt{Esc}\rangle \in \sum$ *be an interaction state. Then, the transition function* $\texttt{next\_AS} \in [\sum \rightarrow \mathcal{P}(\sum \times \texttt{Label\_AS})]$ *is defined as:*

$$\texttt{next\_AS(st)} = \texttt{next\_AS}_{\texttt{dir}}\texttt{(st)} \cup \texttt{next\_AS}_{\texttt{ind}}\texttt{(st)}$$

the semantics of an object is the set of the traces encoding all the interactions between the object and any possible contexts it can be instantiated in. Using the transition function, this semantics is defined as follows:

Definition 18. *(AsmL Object:* $\mathbb{O}_{AS}[\![\texttt{o\_AS}]\!]$*)*
Let $v_{as} \in \texttt{Val}$ *be an AsmL object input value and* $s_{as} \in \texttt{AS\_Store}$ *a store at object creation time. Then the AsmL object semantics,* $\mathbb{O}_{AS}[\![\texttt{o\_as}]\!] \in [D_{\texttt{in}} \times \texttt{AS\_Store}] \rightarrow \mathcal{P}(\mathcal{T}(\Sigma))$ *is defined as:*

$$\mathbb{O}_{AS}[\![\texttt{o\_as}]\!](v_{as}, s_{as}) = lfp_{\emptyset}^{\subseteq}\ \lambda T.\ S_{O}\langle v_{as}, s_{as}\rangle \cup \{\sigma_0 \xrightarrow{l_0} \ldots \xrightarrow{l_{n-1}} \sigma_n \xrightarrow{l_n} \sigma' \mid \{\sigma_0 \xrightarrow{l_0} \ldots \xrightarrow{l_{n-1}} \sigma_n \in T,\ \texttt{next}_{AS}(\sigma_n) \ni \langle \sigma', l'\rangle\}$$

where $D_{\texttt{in}}$ *and* $D_{\texttt{out}}$ *is the semantic domain for the input and output values,* $\sum = \texttt{AS\_Env} \times \texttt{AS\_Store} \times D_{\texttt{out}} \times \mathcal{P}(\texttt{Addr})$ *is a set of interaction states,* $\texttt{next}_{\texttt{as}}(\sigma)$*.*

Using Definition 18, the partial traces semantics of an object can be expressed as a fixpoint:

Theorem 1. *Let*

$$H_{as} = \lambda T.\ S_{O}\langle v_{as}, s_{as}\rangle \cup \{\sigma_0 \xrightarrow{l_0} \ldots \xrightarrow{l_{n-1}} \sigma_n \xrightarrow{l_n} \sigma' \mid \{\sigma_0 \xrightarrow{l_0} \ldots \xrightarrow{l_{n-1}} \sigma_n \in T,\ \texttt{next}_{\texttt{as}}(\sigma_n) \ni \langle \sigma', l'\rangle\}$$

Then $\mathbb{O}_{AS}[\![\texttt{o\_as}]\!](v_{as}, s_{as}) = \cup_{n=0}^{\omega} {H_{as}}^{n}(\emptyset)$

Proof. The proof is immediate from the fixpoint theorem in [3].

As a class is a description of a set of objects, it is natural to define the semantics of a class C as the set that contains the semantics of all the objects that are instances of C. This is expressed by the next definition:

A class is defined as a description of a set of objects. Therefore, its semantics is no more than the semantics of its objects instances according to the following definition:

Definition 19. *(AsmL Class Semantics: $\mathbb{C}_{AS}[\![\texttt{c\_as}]\!]$)*
Let $\texttt{c\_as} = \langle \texttt{as\_dmem, as\_mth, as\_ctr} \rangle$ *be an AsmL class. The semantics of* $\mathbb{C}_{AS}[\![\texttt{c\_as}]\!] \in \mathcal{P}(\mathcal{T}(\Sigma))$ *is:*

$$\mathbb{C}_{as}[\![\texttt{c\_as}]\!] = \{ \mathbb{O}_{AS}[\![\texttt{o\_as}]\!](v_{as}, s_{as}) \mid \texttt{o\_as} \text{ is an instance of } \texttt{c\_as}, \\ \texttt{v\_as} \in \texttt{D\_in},\ \texttt{s\_as} \in \texttt{AS\_Store} \}$$

Using the class semantics in fixpoint in Definition 19, the semantics of each instance takes into account, among others, the interaction with other objects. Therefore, it is possible to merge together all the semantics of the different instances while preserving the program behavior.

Theorem 2. *(AsmL Class semantics in fixpoint) Let*

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

Then $\mathbb{C}_{AS}[\![\texttt{c\_as}]\!](v_{as}, s_{as}) = \textit{lfp}^{\subseteq}_{\emptyset}\ H_{as}\langle\ D_{in} \times \texttt{Store} \rangle$

Proof. Although the AsmL model presents some additional functionalities on top of generic OO languages, the proof of this theorem is similar to the proof of Theorem 3.2 in [9]. For instance, considering the definition of $\mathbb{C}_{AS}$ and applying in order Definition 18, Theorem 1 and the fixpoint theorem in [3], the proof is straightforward.

4 Soundness and Correctness of the Class Semantics

The last step in the AsmL fixpoint semantics is to relate the classes semantics to the whole AsmL program semantics. For this purpose, we consider updated versions of the functions *split* ($\alpha^{\circ}_{✂}$), *project* ($\alpha^{\circ}_{\uparrow}$) and *abstract* (α°) as defined in [9]. The new functions do support the AsmL update semantics, environment and store.

The basic concept behind defining the object semantics is to cut all the instances not involving the object. For this purpose, two helper functions are required: (1) $\alpha\texttt{\_AS}^{\circ}_{✂}$ that cuts all the traces involving the object instances; and (2) $\alpha\texttt{\_AS}^{\circ}_{\uparrow}$ that maps all the cut instances to interaction states.

Let us first define the helper function `split_AS`, which given a trace τ and an object `o_as`, it returns a pair consisting of the last state of the prefix of τ made up of the last state of the execution of a method or process of `o_as` and the remaining suffix of prefix of τ.

Definition 20. *(The split helper function* `split_AS`*)*
Let `o_as` *be an AsmL object,* $\tau \in \mathcal{T}(\texttt{AS\_Env} \times \texttt{AS\_Store})$*,* $\texttt{CurMethod} \in \texttt{Mth}$

and $\mathtt{pc}_{exit}$ be the exit point of $\tau(0)(\mathtt{CurMethod})$. Then $\mathtt{split\_AS} \in [(\mathcal{T}(\mathtt{AS\_Env}\times \mathtt{AS\_Store}) \rightarrow (\mathtt{AS\_Env}\times \mathtt{AS\_Store}) \times \mathcal{T}(\mathtt{AS\_Env} \times \mathtt{AS\_Store})]$ is defined as:

$$\begin{array}{lll} \mathtt{split\_AS}(\tau) = & \mathit{let}\ n = \mathit{min}\{ & i \in \mathbb{N} \mid \tau(i) = \langle \mathtt{CurMethod} \rangle \\ & & \wedge\ \tau(i)(\mathtt{SL}) = \mathtt{true} \wedge \tau(i)(\mathtt{pc}) = \mathtt{pc}_{exit} \wedge \\ & & \tau(i)(\mathtt{this}) = \mathtt{o\_as} \wedge \tau(i)(\mathtt{StackHeight}) = \\ & & \tau(0)(\mathtt{StackHeight})\} \\ & \mathit{in} & \langle \tau(n), \tau(n+1) \rightarrow \ldots \rightarrow \tau(\mathtt{Len}(\tau)-1) \rangle \end{array}$$

The cut function, $\alpha\_\mathtt{AS}^{\circ}_{\text{✂}}$, cuts all the sub-traces of that do not involve a specific object. It considers four different cases:

1. For an empty trace, ϵ, it returns an empty trace.
2. If trace is part of the object trace, then we split it recursively keeping only the last state of the execution of a method or process. The rest of the trace is removed.
3. If the trace belongs to an object different from the current object and the store is not changed, then we continue with the rest of the trace.
4. If the trace belongs to an object different from the current object and the store is changed, then we keep the current trace and we continue with the rest of the traces.

Definition 21. *(Cut function: $\alpha\_\mathtt{AS}^{\circ}_{\text{✂}}$)*
Let $\mathtt{o\_as}$ *be an AsmL object,* $\tau \in \mathcal{T}(\mathtt{AS\_Env}\times \mathtt{AS\_Store})$. *Then* $\alpha\_\mathtt{AS}^{\circ}_{\text{✂}} \in [(\mathcal{T}(\mathtt{AS\_Env}\times \mathtt{AS\_Store}) \times \mathtt{AS\_Store}) \rightarrow \mathcal{T}(\mathtt{AS\_Env}\times \mathtt{AS\_Store})]$ *is defined as:*

$$\alpha\_\mathtt{AS}^{\circ}_{\text{✂}} = \lambda(\tau, S_{\mathtt{last}}).$$

$$\begin{cases} \epsilon & \mathtt{if}\ \tau = \epsilon \\ \begin{array}{l} \mathtt{let}\ \langle \rho', \tau' \rangle = \mathtt{split\_AS}(\tau) \\ \mathtt{in}\ \mathtt{let}\ \langle e'_{as}, s'_{as} \rangle = \rho' \\ \mathtt{in}\ \rho' \rightarrow \alpha\_\mathtt{AS}^{\circ}_{\text{✂}}(\tau', s'_{as}) \end{array} & \mathtt{if}\ \tau = \langle e_{as}, s_{as} \rangle \rightarrow \tau'', e_{as}(\mathtt{this}) = \mathtt{o\_as} \\ \alpha\_\mathtt{AS}^{\circ}_{\text{✂}}(\tau'', S_{\mathtt{last}}) & \begin{array}{l} \mathtt{if}\ \tau = \langle e_{as}, s_{as} \rangle \rightarrow \tau'', \\ e_{as}(\mathtt{this}) \neq \mathtt{o\_as}, \\ S_{/S(\mathtt{o\_as})} = S_{\mathtt{last}/S(\mathtt{o\_as})} \end{array} \\ \langle e_{as}, s_{as} \rangle \rightarrow \alpha\_\mathtt{AS}^{\circ}_{\text{✂}}(\tau'', S) & \begin{array}{l} \mathtt{if}\ \tau = \langle e_{as}, s_{as} \rangle \rightarrow \tau'', \\ e_{as}(\mathtt{this}) \neq \mathtt{o\_as}, \\ S_{/S(\mathtt{o\_as})} \neq S_{\mathtt{last}/S(\mathtt{o\_as})} \end{array} \end{cases}$$

The second part of the abstraction includes the $\alpha\_\mathtt{AS}^{\circ}_{\uparrow}$ function which maps the states of a trace to interaction states. It considers three different cases:

1. For empty trace, ϵ, it returns an empty trace.

2. If the trace is part of the object trace, it returns the set of direct interaction states.
3. If the trace belongs to an object different from the current object, it returns the indirect interaction states..

Definition 22. *(Map function: $\alpha\_AS^{\circ}_{\uparrow}$)*
Let $\texttt{o\_as}$ *be an AsmL object,* $\tau \in \mathcal{T}(\texttt{AS\_Env}\times \texttt{AS\_Store})$*. Then* $\alpha\_AS^{\circ}_{\uparrow} \in [(\mathcal{T}(\texttt{AS\_Env}\times \texttt{AS\_Store}) \times \mathcal{P}(\texttt{Addr})) \to \mathcal{T}(\sum)]$ *is defined as:*

$$\alpha\_AS^{\circ}_{\uparrow} = \lambda(\tau, \mathit{Esc}).$$

$$\begin{cases} \epsilon & \mathit{if}\ \ \tau = \epsilon \\ \\ \mathit{let}\ \langle e_{as}, s_{as}\rangle = \rho \\ \mathit{in}\ \mathit{let}\ \mathit{Esc}' = \mathit{Esc} \cup \\ \quad \mathit{reachable\_AS}(\rho(\mathit{retVal}), s_{as}) & \mathit{if}\ \ \tau = \rho \to \tau', e_{as}(\mathit{this}) = \texttt{o\_as} \\ \mathit{in}\ \langle\langle e_{as}, s_{as}, \rho(\mathit{retVal}), \mathit{Esc}\rangle, \\ \langle \rho(\mathit{curMethod}), \rho(\mathit{inVal})\rangle\rangle \\ \to \alpha\_AS^{\circ}_{\uparrow}(\tau', \mathit{Esc}') \\ \\ \mathit{let}\ \langle e_{as}, s_{as}\rangle = \rho \\ \mathit{in}\ \langle\langle e_{as}, s_{as}, \phi, \mathit{Esc}\rangle, k\rangle & \mathit{if}\ \ \tau = \rho \to \tau', e_{as}(\mathit{this}) \neq \texttt{o\_as} \\ \to \alpha\_AS^{\circ}_{\uparrow}(\tau', \mathit{Esc}) \end{cases}$$

The abstraction function $\alpha\_\texttt{AS}^{\circ}$ projects from the traces of an execution of the set of relevant states to a specific object.

Definition 23. *(Abstract function: α_AS°)*
Let $\texttt{o\_as}$ *be an AsmL object,* $T \subseteq \mathcal{T}(\texttt{AS\_Env}\times \texttt{AS\_Store})$ *a set of execution traces and* $s_{\emptyset}$ *the empty store. The abstraction function* $\alpha\_AS^{\circ} \in [(\mathcal{T}(\texttt{AS\_Env}\times \texttt{AS\_Store}) \to \mathcal{P}(\mathcal{T}(\sum))]$ *is defined as:*

$$\alpha\_AS^{\circ}(T) = \{\alpha\_AS^{\circ}_{\uparrow}(\alpha\_AS^{\circ}_{\times}(\tau, s_{\emptyset}), \emptyset) \mid \tau \in T\}$$

We can now state the relation between the whole AsmL program semantics $\mathbb{W}[\![\texttt{AS_Pg}]\!]$and the AsmL class semantics $\mathbb{C}_{AS}[\![\texttt{c\_as}]\!]$. The following theorem states that all interactions that a program `AS_Pg` performs on an object `o_AS` an instance of a class `C_AS` are captured by $\mathbb{C}_{AS}[\![\texttt{c\_as}]\!]$.

Theorem 3. *(Soundness of $\mathbb{C}_{AS}[\![\texttt{c\_as}]\!]$) Let* P_{AS} *be a whole AsmL program and let* $\texttt{c}_{AS} \in C_{AS}$*. Then*

$\forall\ R_0 \in \texttt{AS\_Env}\times \texttt{AS\_Store}.\ \forall\ \tau \in \mathcal{T}(\texttt{AS\_Env}\times \texttt{AS\_Store}).\ \tau \in \mathbb{W}[\![\texttt{AS\_Pg}]\!](R_0) \Rightarrow \exists \tau' \in \mathbb{C}_{AS}[\![\texttt{c}_{AS}]\!].\ \alpha\_AS^{\circ}(\{\tau\}) = \{\tau'\}$

Proof. (Sketch) We have to consider both cases when τ contains an object $\texttt{o}_{AS}$, instantiation of $\texttt{m}_{AS}$, and when it does not include any $\texttt{o}_{AS}$. 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 AsmL classes are initialized in the main program `Main` before the execution starts, there exist an initial environment, store and set of variables that define the initial trace $\sigma_0 \in \tau''$. The rest of the traces in τ'' are interaction states of $\mathtt{o}_{AS}$ because they are obtained by applying $\alpha\_\mathtt{AS}^{\circ}$ on τ. Therefore, $\tau'' \in \mathbb{C}_{AS}[\![\mathtt{m}_{AS}]\!]$.

The following theorem states that all the behaviors considered by $\mathbb{C}_{AS}[\![\mathtt{c_as}]\!]$are feasible. In other terms, for any AsmL class, there exists an AsmL program that will contain all the possible behaviors of this class.

Theorem 4. *(Completeness of $\mathbb{C}_{AS}[\![]\!]$) Let $\mathtt{c}_{AS}$ be an AsmL class. Then*

$$\forall \tau \in \mathcal{T}(\Sigma).\ \tau \in \mathbb{C}_{AS}[\![\mathtt{c}_{AS}\ ]\!] \Rightarrow \exists\ \mathit{AS\_P} \in \langle L_{\mathit{AS\_Pg}} \rangle.\ \exists \rho_0 \in \mathit{AS\_Env} \times \mathit{AS\_Store}.\ \exists\ \mathtt{o}_{AS}\ \textit{instance of}\ \mathtt{c}_{AS}.\ \textit{exists}\ \tau' \in \mathcal{T}(\mathit{AS\_Env} \times \mathit{AS\_Store}).\ \tau' \in \mathbb{W}[\![\rho_0]\!] \wedge \alpha\_\mathit{AS}^{\circ}(\{\tau'\}) = \{\tau\}$$

Proof. An AsmL program satisfying the previous theorem can be constructed by creating and instance of $\mathtt{c}_{AS}$ in the `Main` function, the initial state corresponds to the state when the class constructor, `AS_Ctr`, was executed. It is always possible to construct both `AS_P` and ρ_0. For instance, there exist many other possible constructions involving AsmL methods pre-conditions and post-conditions.

5 Application

In this Section, we provide a direct application of the proposed AsmL semantics, namely to prove the soundness of the transformation of AsmL to SystemC [10] and vice-versa. The SystemC language 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 SystemC designs is not feasible due to the complexity of the SystemC library and its simulator. To solve this problem, we proposed in [6] to translate SystemC models to an intermediate representation in AsmL more suitable for formal verification. This approach reduced radically the complexity of the design at the point that we were able to verify a complete PCI architecture using the SMV model checker [11].

In our verification methodology of Figure 1, we perform the model checking of SystemC by translating the original design to an abstract representation omitting completely the details of the SystemC simulator. The target (or transformed) program is modeled in AsmL to be cross-produced with the system property and checked over the whole system's state space. We embed the PSL (Property Specification Language) [1] properties in the design as external monitors; hence, these monitors can be used as stand-alone blocks to validate other devices, either at the AsmL level by model checking or at the SystemC level by ABV. To ensure the correctness of the model checking results at the AsmL level on the original SystemC program, we defined a set of rules that translate the code from SystemC to AsmL. These rules represented an informal representation of the soundness of the approach.

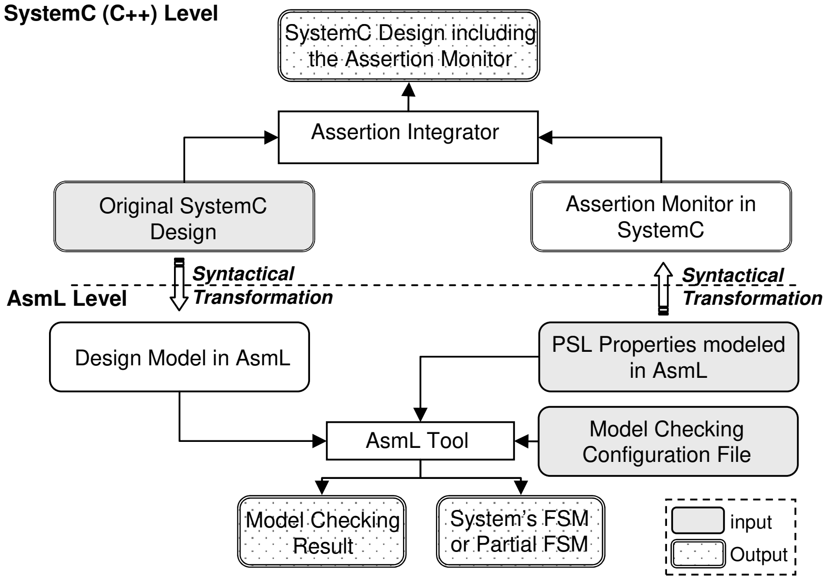

Fig. 1. Verification Methodology.

By providing a formalization of the SystemC and AsmL semantics in fixpoint based on the OO general case given in [9], we proved in [7] 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 transformation frameworks defined in [4]. Such a result will enable using a variety of formal tools (for e.g., SMV for model checking [11]) or to use AsmL tool (Asmlt) to generate a finite state machine of the design.

6 Conclusion

In this paper, we presented the fixpoint semantics of the abstract state machines language (AsmL). Then, we proved the soundness and the correctness of the the semantics of the AsmL class AS_Class w.r.t. to a trace semantics of a the whole AsmL program. Such a result presents a first step towards applying formal methods to AsmL. For instance, we used these semantics to prove the soundness of a transformation from AsmL to SystemC, which enabled verifying SystemC transactional models. Furthermore, the concrete semantics, we defined, can be used to construct sound abstract semantics allowing static code analysis or model chcking of AsmL programs.

References

1. Accellera Organization. Accellera property specification language reference manual, version 1.01. www.accellera.org, 2004.
2. P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theor. Comput. Sci.*, 277(1-2):47–103, 2002.
3. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, USA, 1979.
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, Portland, Oregon, January 2002.
5. Y. Gurevich, B. Rossman, and W. Schulte. Semantic Essence of AsmL. Technical report, Microsoft Research Tech. Report MSR-TR-2004-27, March 2004.
6. A. Habibi and S. Tahar. Design for verification of SystemC transaction level models. In *Proc. Design Automation and Test in Europe*, Munich, Germany, March 2005.
7. A. Habibi and S. Tahar. On the transformation of SystemC to AsmL using abstract interpretation. *Electr. Notes Theor. Comput. Sci.*, 2005 (to appear).
8. S. N. Kamin and U. S. Reddy. Two semantic models of object-oriented languages. pages 463–495, 1994.
9. F. Logozzo. *Anhalyse Statique Modulaire de Langages a Objets*. PhD thesis, Ecole Polytechnique, Paris, France, June 2004.
10. Open SystemC Initiative. www.systemc.org, 2004.
11. K. Oumalou, A. Habibi, and S. Tahar. Design for verification of a PCI bus in SystemC. In *Proc. Symposium on System-on-Chip*, pages 201–204, Tampere, Finland, November 2004.