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. They are 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 [9]. 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 expends on the generic semantics provided by Logozzo in [10] 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.

**Definition 1. (AsmL Class: AS\textsubscript{C})**

An AsmL class is a set \( \langle \text{AS}_{\text{DMem}}, \text{AS}_{\text{Mth}}, \text{AS}_{\text{Ctr}} \rangle \), where \( \text{AS}_{\text{DMem}} \) is a set of the class data members, \( \text{AS}_{\text{Mth}} \) a set of methods (functions) definition and \( \text{AS}_{\text{Ctr}} \) 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).

**Definition 2. (AsmL Method: AS\textsubscript{Mth})**

An AsmL method is a set \( \langle \text{AS}_{\text{M}}, \text{AS}_{\text{Pre}}, \text{AS}_{\text{Pos}}, \text{AS}_{\text{Cst}} \rangle \), where \( \text{AS}_{\text{M}} \) is a the core of the method, \( \text{AS}_{\text{Pre}} \) is a set of pre-conditions, \( \text{AS}_{\text{Pos}} \) is a set of post-conditions and \( \text{AS}_{\text{Cst}} \) is a set of constraints.
Note that \textit{AS\textsubscript{Pre}}, \textit{AS\textsubscript{Pos}} and \textit{AS\textsubscript{Cst}} share the same structure. They are differentiated in the methods by using a specific keyword for each of them (e.g., \textit{require} for pre-conditions).

\textbf{Definition 3. (AsmL Method Precondition: \textit{AS\textsubscript{Pre}})}
An AsmL method pre-condition is a set \{\textit{AS\textsubscript{B}}\}, where \textit{AS\textsubscript{B}} is a Boolean proposition.

\textbf{Definition 4. (AsmL Program: \textit{AS\textsubscript{Pg}})}
An AsmL Program is a set \{\textit{LAS\textsubscript{C}}, \textit{INIT}\}, where \textit{LAS\textsubscript{C}} is a set of AsmL classes and \textit{INIT} is the main function in the program.

\subsection{2.2 Semantical Domains}
AsmL considers two phases: \textit{evaluate} and \textit{update}. The program will be always running in the \textit{evaluate} mode except if an update is requested. There are two types of updates, total and partial.

\textbf{Definition 5. (Total Update: \textit{Step})}
A total update is performed using the \textit{Step} instruction and affects all the program's variables.

\textbf{Definition 6. (AsmL Environment: \textit{AS\textsubscript{Env}})}
The AsmL Environment is a modified OO environment \textit{AS\textsubscript{Env}} = \{\textit{Var} \rightarrow \textit{Addr}, \textit{Addr}\}, where \textit{Var} is a set of variables and \textit{Addr} \subseteq \mathbb{N} is a set of addresses.

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

\textbf{Definition 7. (AsmL Store: \textit{AS\textsubscript{Store}})}
The AsmL store is \textit{AS\textsubscript{Store}} = \{\textit{Addr}, \textit{Addr}\} \rightarrow \{\textit{Val}, \textit{Val}\}, where \textit{Val} is a set of values such that \textit{AS\textsubscript{Env}} \subseteq \textit{Val}.

Let \textbf{R} \in \mathcal{P} (\textit{AS\textsubscript{Env}} \times \textit{AS\textsubscript{Store}}) be a set of initial states, \textit{pcin} be the entry point of the main function \textit{Main} and \rightarrow \subseteq (\textit{AS\textsubscript{Env}} \times \textit{AS\textsubscript{Store}}) \times (\textit{AS\textsubscript{Env}} \times \textit{AS\textsubscript{Store}}) be a transition relation.

\section{3 Fixpoint Semantics}
\subsection{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 \textbf{R} \textsubscript{0}. It can be expressed in fixpoint semantics as follows:

\textbf{Definition 8. (Whole AsmL Program Semantics: \textit{\mathcal{W}_AS [\textit{AS\textsubscript{Pg}}]})}
Let \textit{AS\textsubscript{Pg}} = \{\textit{LAS\textsubscript{C}}, \textit{Main}\} be an AsmL program. Then, the semantics of \textit{AS\textsubscript{Pg}}, \textit{\mathcal{W}_AS [\textit{AS\textsubscript{Pg}}] \subseteq \mathcal{P}(\textit{AS\textsubscript{Env}} \times \textit{AS\textsubscript{Store}})} → \mathcal{P}(\textit{AS\textsubscript{Env}} \times \textit{AS\textsubscript{Store}})) is
\[
\textit{\mathcal{W}_AS [\textit{AS\textsubscript{Pg}}]}(\textbf{R} \textsubscript{0}) = \mathrm{lfp} \subseteq \lambda \textit{X}. (\textbf{R} \textsubscript{0} \cup \{\rho_0 \rightarrow \ldots \rho_n \rightarrow \rho_{n+1}\} | \rho_{n+1} \in \{\textit{AS\textsubscript{Env}} \times \textit{AS\textsubscript{Store}}\} \wedge \{\rho_0 \rightarrow \ldots \rho_n\} \in \textit{X} \wedge \rho_n \rightarrow \rho_{n+1})
\]
Definition 9. (Method Semantics: $\mathcal{M}_\text{AS}$)
Let $\text{AS}_\text{Mth} = (\text{AS}_\text{M}, \text{AS}_\text{Pre}, \text{AS}_\text{Pos}, \text{AS}_\text{Cst})$ be an AsmL method. Then, the semantics of $\text{AS}_\text{Mth}$, $\mathcal{M}_\text{AS} [\text{AS}_\text{M}]$ be an AsmL method. Then, the semantics of $\text{AS}_\text{Mth}$, $\mathcal{M}_\text{AS} [\text{AS}_\text{M}]$ is:

$$
\mathcal{M}_\text{AS} [\text{AS}_\text{M}](\text{R}_0, \text{M}, \text{Pre}, \text{Pos}, \text{Cst}) = \text{lfp} \subseteq \emptyset \lambda X, m, \text{pre}, \text{pos}, \text{cst}. \{\text{R}_0 \cap \{\rho_0 \rightarrow \ldots \rightarrow \rho_n \rightarrow \rho_{n+1} | \rho_{n+1} \in \langle \text{AS}_\text{Env} \times \text{AS}_\text{Store} \rangle \wedge \{\rho_0 \rightarrow \ldots \rightarrow \rho_n \} \in X \wedge \rho_n \rightarrow \rho_{n+1} \wedge \rho_{n+1}(X) = (m, \text{pre}, \text{pos}, \text{cst}) \wedge \text{pre} = \text{pos} = \text{cst} = \text{true}\}.
$$

3.2 AsmL Class Semantics

The AsmL class constructor is a default OO constructor. It can be defined according to the Definition 3.8 in [10].

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 [10]. 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 [10], this interaction was very well defined using two functions $\text{next}_d$, for direct interactions, and $\text{next}_\text{ind}$ for indirect interactions and the object semantics, $\mathcal{O} [\sigma]$, was defined as:

$$
\mathcal{O} [\sigma](v, s) = \text{lfp} \subseteq \lambda T. \mathcal{S}_0(v, s) \cup \{\sigma_0 \overset{l_0}{\rightarrow} \ldots \overset{l_{n-1}}{\rightarrow} \sigma_n \overset{l_n}{\rightarrow} \sigma’| \\{\sigma_0 \overset{l_0}{\rightarrow} \ldots \overset{l_{n-1}}{\rightarrow} \sigma_n \in T, \text{next}(\sigma_n) \ni \{\sigma’, l’\}\}
$$

where $\text{next}(\sigma) = \text{next}_d(\sigma) \cup \text{next}_\text{ind}(\sigma)$, $l$ is a transition label and $\mathcal{S}_0(v, s)$ is a set of initial states.

In addition to the semantics definition of an OO object in [10], an AsmL method can be activated by an update instruction. This interaction is a hybrid direct/indirect interaction because, according the state of the program events (that maybe external to the object), invoke directly the concerned methods. In following, we will define the interaction states, then, we will provide the complete definition for the direct, indirect and AsmL specific interaction functions.

Definition 10. (Interaction States)
The set of interaction states is $\sum = \text{AS}_\text{Env} \times \text{AS}_\text{Store} \times \text{D}_\text{out} \times \mathcal{P}(\text{Addr})$.

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

Definition 11. (Initial States $\mathcal{S}_0(v_{\text{as}}, s_{\text{as}})$)
Let $v_{\text{as}} \in \text{D}_\text{in}$ be an AsmL object input value, $s_{\text{as}} \in \text{AS}_\text{Store}$ a store at object creation time and $\text{AS}_\text{Obj}$ an AsmL object. The set of initial states of $\text{AS}_\text{Obj}$ is:

$$
\mathcal{S}_0(v_{\text{as}}, s_{\text{as}}) = \{ (e_{\text{as}}, s_{\text{as}}, \phi, \emptyset) | \mathcal{P}_\text{Ctr}[\text{AS}_\text{Ctr}](L_M) \ni (e_{\text{as}}, s_{\text{as}}) \}
$$

where: $L_M = \{m_1, \ldots, m_i, \ldots, m_n\}$ is a list of the the methods.
In Definition 11 φ is a void value (∈ D_{out}) meaning that the constructor does not return any value and therefore does not expose any address to the context.

Definition 12. (Transition Labels: Label_{AS})
The set of transition labels is \( \text{Label}_{AS} = (\text{Mth} \times \text{D}_{in}) \cup \{k\} \).

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: next_{AS_{dir}} and next_{AS_{ind}}.

Definition 13. (Direct interactions: next_{AS_{dir}})
Let \( \langle e_{as}, s_{as}, v_{as}, \text{Esc} \rangle \in \Sigma \) an interaction state. Then, the direct interaction function next_{AS_{dir}} ∈ \[\Sigma \rightarrow \mathcal{P}(\Sigma \times \text{Label}_{AS})\] is defined as:

\[
\text{next}_{AS_{dir}}(\langle e_{as}, s_{as}, v_{as}, \text{Esc} \rangle) =
\{ \langle e'_{as}, s'_{as}, \text{Esc}' \rangle \mid \exists m_{th} \in \text{Mth}, v_{in} \in \text{D}_{in}, \ C[m_{th}](v_{in}, e_{as}, s_{as}) \}.
\]

where \( C[m_{th}] \) is the semantics of generic OO method as defined in [10].

The function reachable is an extension of the helper function of the one defined in [10]. 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 ∈ \([\text{D}_{out} \times \text{AS}_{Store}] \rightarrow \mathcal{P}(\text{Addr})\) is defined as follows:

\[
\text{reachable}(v_{as}, s_{as}) =
\begin{cases}
\text{Addr} & \text{if } v_{as} \in \text{Addr} \text{ then } \\
\{ \text{Addr} \} \cup \{ \text{reachable}(e'_{as}(d_{mem}), s'_{as}) \mid \exists \text{as}_{class} = \langle \text{AS}_{DMem}, \text{AS}_{Mth}, \text{AS}_{Ctr} \rangle, d_{mem} \in \text{AS}_{DMem}, s_{as}(v_{as}) \text{ is an instance of as}_{class}, s_{as}(s_{as}(v_{as})) = e'_{as} \} & \text{else } 0.
\end{cases}
\]

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 next_{AS_{ind}} defines this type of interaction:

Definition 15. (Indirect interactions: next_{AS_{ind}})
Let \( \langle e_{as}, s_{as}, v_{as}, \text{Esc} \rangle \in \Sigma \) an interaction state. Then, the indirect interaction function next_{AS_{ind}} ∈ \[\Sigma \rightarrow \mathcal{P}(\Sigma \times \text{Label}_{AS})\] is defined as:

\[
\text{next}_{AS_{ind}}(\langle e_{as}, s_{as}, v_{as}, \text{Esc} \rangle) =
\{ \langle e_{as}, \phi, \text{Esc}, k \rangle \mid \exists \alpha \in \text{Esc}, s'_{as} \in \text{update}_{as}(\alpha, s_{as}) \}.
\]
The update\_as function is an extension of the update function defined in [10] 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 $\text{update}\_\text{as} \in [\text{Addr} \times \text{AS}\text{Store} \rightarrow \mathcal{P}(\text{AS}\text{Store})]$ is defined as follows:

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

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

Using the definitions of $\text{next}\_\text{ASdir}$ and $\text{next}\_\text{ASind}$, we define the global transition function $\text{next}\_\text{AS}$ as:

**Definition 17. (Transition function: next\_AS)**
Let $st = (c_{\text{as}}, s_{\text{as}}, v_{\text{as}}, \text{Esc}) \in \Sigma$ be an interaction state. Then, the transition function $\text{next}\_\text{AS} \in [\Sigma 

\rightarrow \mathcal{P}(\Sigma \times \text{Label}\_\text{AS})]$ is defined as:

$\text{next}\_\text{AS}(st) = \text{next}\_\text{ASdir}(st) \cup \text{next}\_\text{ASind}(st)$

Using the transition function, an AsmL object semantics is defined as follows:

**Definition 18. (AsmL Object: $\text{O}_{\text{AS}}[c_{\text{AS}}]$)**
Let $v_{\text{as}} \in \text{Val}$ be an AsmL object input value and $s_{\text{as}} \in \text{AS}\text{Store}$ a store at object creation time. Then the AsmL object semantics, $\text{O}_{\text{AS}}[c_{\text{AS}}] \in [\text{D}\text{in} \times \text{AS}\text{Store} \rightarrow \mathcal{P}(T(\Sigma))]$ is defined as:

$\text{O}_{\text{AS}}[c_{\text{AS}}](v_{\text{as}}, s_{\text{as}}) = \{l \in \Sigma . S_0(v_{\text{as}}, s_{\text{as}}) \cup \{\sigma_0 \xrightarrow{l_0} \ldots \xrightarrow{l_{n-1}} \sigma_n \xrightarrow{l_n} \sigma'\} \mid \{\sigma_0 \xrightarrow{l_0} \ldots \xrightarrow{l_{n-1}} \sigma_n \in T, \text{next}\_\text{AS}(\sigma_n) \ni \langle \sigma', l' \rangle\}$

where $\text{D}\text{in}$ and $\text{D}\text{out}$ is the semantic domain for the input and output values, $\Sigma = \text{AS}\text{Env} \times \text{AS}\text{Store} \times \text{D}\text{out} \times \mathcal{P}(\text{Addr})$ is a set of interaction states, $\text{next}\_\text{AS}(\sigma)$. The detailed definition of $\text{next}\_\text{AS}$ can be found in [6].

**Theorem 1.** Let

$F_{\text{as}} = \lambda T. S_0(v_{\text{as}}, s_{\text{as}}) \cup \{\sigma_0 \xrightarrow{l_0} \ldots \xrightarrow{l_{n-1}} \sigma_n \xrightarrow{l_n} \sigma'\} \ni \{\sigma_0 \xrightarrow{l_0} \ldots \xrightarrow{l_{n-1}} \sigma_n \in T, \text{next}\_\text{AS}(\sigma_n) \ni \langle \sigma', l' \rangle\}$

Then $\text{O}_{\text{AS}}[c_{\text{AS}}](v_{\text{as}}, s_{\text{as}}) = \bigcup_{n=0}^{\infty} F_{\text{as}}^n(\emptyset)$

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

**Definition 19. (AsmL Class Semantics: $\text{C}_{\text{AS}}[c_{\text{AS}}]$)**
Let $c_{\text{as}} = (\text{as}\_\text{dmem}, \text{as}\_\text{mth}, \text{as}\_\text{ctr})$ be an AsmL class. The semantics of $\text{C}_{\text{AS}}[c_{\text{AS}}] \in \mathcal{P}(T(\Sigma))$ is:

$\text{C}_{\text{AS}}[c_{\text{AS}}] = \{\text{O}_{\text{AS}}[c_{\text{AS}}](v_{\text{as}}, s_{\text{as}}) \mid c_{\text{as}} \text{ is an instance of } c_{\text{as}}, v_{\text{as}} \in \text{D}\text{in}, s_{\text{as}} \in \text{AS}\text{Store}\}$
Theorem 2. (AsmL Class semantics in fixpoint) Let
\[ H_{as}(S) = \lambda T. \{ (v, s) \mid (v, s) \in S \} \cup \{ \sigma_0 \xrightarrow{l_0} \ldots \xrightarrow{l_{n-1}} \sigma_n \xrightarrow{l'} \sigma' \mid \sigma_0 \xrightarrow{l_0} \ldots \xrightarrow{l_{n-1}} \sigma_n \in T, \text{nextas}(\sigma_n) \supset (\sigma', l') \} \]

Then \( C_{AS} \circ_{vas} (v_{as}, s_{as}) = \text{lfp} \subseteq H_{as}(D_{in} \times \text{Store}) \)

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 [10]. For instance, considering the definition of \( C_{AS} \) 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 \( \text{split} (\alpha \circ q) \), \( \text{project} (\alpha \circ \uparrow) \) and \( \text{abstract} (\alpha \circ) \) as defined in [10]. The new functions are upgraded to 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) \( \alpha_{AS} \circ q \) that cuts all the traces involving the object instances; and (2) \( \alpha_{AS} \circ \uparrow \) that maps all the cut instances to interaction states.

Let us first define the helper function \( \text{split}_{AS} \), which given a trace \( \tau \) and an object \( o_{as} \), it returns a pair consisting of the last state of the execution of a method or process of \( o_{as} \) and the remaining suffix of prefix of \( \tau \).

Definition 20. (The split helper function \( \text{split}_{AS} \))

Let \( o_{as} \) be an AsmL object, \( \tau \in T(AS_{En} \times AS_{Store}) \), \( \text{CurMethod} \in \text{Mth} \) and \( pc_{exit} \) be the exit point of \( \tau(0)(\text{CurMethod}) \). Then \( \text{split}_{AS} \in [(T(AS_{En} \times AS_{Store}) \times T(AS_{En} \times AS_{Store})) \rightarrow (AS_{En} \times AS_{Store}) \times T(AS_{En} \times AS_{Store})] \) is defined as:

\[
\text{split}_{AS}(\tau) = \text{let } n = \min \{ i \in \mathbb{N} \mid \tau(i) = (\text{CurMethod}) \\
\land \tau(i)(\text{SL}) = \text{true} \land \tau(i)(pc) = pc_{exit} \land \\
\tau(i)(\text{this}) = o_{as} \land \tau(i)(\text{StackHeight}) = \\
\tau(0)(\text{StackHeight}) \}
\text{in } \langle \tau(n), \tau(n+1) \rightarrow \ldots \rightarrow \tau(\text{Len}(\tau) - 1) \rangle
\]

The cut function \( \alpha_{AS} \circ q \) considers four different cases:

1. for empty trace, \( \epsilon \), 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 this is not the current object and the store is not changed, then we continue with the rest of the trace.

4. If this is not 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: \( \alpha_{AS_x} \))**

Let \( o_{as} \) be an AsmL object, \( \tau \in T(AS_{Env} \times AS_{Store}) \). Then \( \alpha_{AS_x} \in \{T(AS_{Env} \times AS_{Store}) \times AS_{Store}) \rightarrow T(AS_{Env} \times AS_{Store}) \} \) is defined as:

\[
\alpha_{AS_x} = \lambda(\tau, S_{\text{last}}).
\]

\[
\left\{ \begin{array}{ll}
\epsilon & \text{if } \tau = \epsilon \\
\text{let } (\rho', \tau') = \text{split}_{AS}(\tau) & \\
\text{in let } (e_{as}, s_{as}) = \rho' & \\
\text{in } \rho' \rightarrow \alpha_{AS_x}(\tau', s_{as}) & \\
\alpha_{AS_x}(\tau'', S_{\text{last}}) & \\
\text{if } \tau = (e_{as}, s_{as}) \rightarrow \tau'', e_{as}(\text{this}) = o_{as} & \\
\text{let } \langle e_{as}, s_{as} \rangle = \rho & \\
\text{if } \tau = (e_{as}, s_{as}) \rightarrow \tau'', e_{as}(\text{this}) \neq o_{as}, & \\
S/S(o_{as}) = S_{\text{last}}/S(o_{as}) & \\
e_{as}(\text{this}) \neq o_{as}, & \\
S/S(o_{as}) \neq S_{\text{last}}/S(o_{as}) & \\
\end{array} \right.
\]

The second part of the abstraction includes the \( \alpha_{AS}^o \) function which maps the states of a trace to interaction states.

**Definition 22. (Map function: \( \alpha_{AS}^o \))**

Let \( o_{as} \) be an AsmL object, \( \tau \in T(AS_{Env} \times AS_{Store}) \). Then \( \alpha_{AS}^o \in \{T(AS_{Env} \times AS_{Store}) \times \mathcal{P}(\text{Addr}) \rightarrow T(\sum) \} \) is defined as:

\[
\alpha_{AS}^o = \lambda(\tau, \text{Esc}).
\]

\[
\left\{ \begin{array}{ll}
\epsilon & \text{if } \tau = \epsilon \\
\text{let } (e_{as}, s_{as}) = \rho & \\
\text{in let } \text{Esc'} = \text{Esc} \cup \text{reachable}_{AS}(\rho(\text{retVal}), s_{as}) & \\
\text{if } \tau = \rho \rightarrow \tau', e_{as}(\text{this}) = o_{as} & \\
\langle e_{as}, s_{as}, \rho(\text{retVal}), \text{Esc} \rangle, \rho(\text{curMethod}), \rho(\text{inVal}) \rangle & \\
\rightarrow \alpha_{AS}^o(\tau', \text{Esc'}) & \\
\text{let } (e_{as}, s_{as}) = \rho & \\
\text{in } \langle e_{as}, s_{as}, \phi, \text{Esc} \rangle, \text{k} & \\
\text{if } \tau = \rho \rightarrow \tau', e_{as}(\text{this}) \neq o_{as} & \\
\rightarrow \alpha_{AS}^o(\tau', \text{Esc}) & \\
\end{array} \right.
\]

The abstraction function \( \alpha_{AS}^o \) projects from the traces of an execution of the set of relevant states to a specific object.
Definition 23. (Abstract function: $\alpha_{AS}$)

Let $o_{AS}$ be an AsmL object, $T \subseteq T(AS_{Env} \times AS_{Store})$ a set of execution traces and $s_0$ the empty store. The abstraction function $\alpha_{AS} \in \mathcal{P}(T(AS_{Env} \times AS_{Store}))$ is defined as:

$$\alpha_{AS}(T) = \{ \alpha_{AS}(o_{AS}, (\tau, s_0)) | \tau \in T \}$$

Theorem 3. (Soundness of $C_{AS}[c_{AS}]$)

Let $P_{AS}$ be a whole AsmL program and let $c_{AS} \in C_{AS}$. Then

$$\forall R_0 \in AS_{Env} \times AS_{Store}. \forall \tau \in T(AS_{Env} \times AS_{Store}). \tau \in W[AS_{Pg}(R_0) : \exists \tau' \in C_{AS}[c_{AS}]. \alpha_{AS}((\tau)) = \{\tau'\}$$

Proof. (Sketch) We have to consider both cases when $\tau$ contains an object $o_{AS}$, instantiation of $m_{AS}$, and when it does not include any $o_{AS}$. For the second situation, the proof of the theorem is trivial considering that $\tau$ will be an empty trace. In the first case, the trace is not empty (let it be $\tau''$). 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 $\tau''$ are interaction states of $o_{AS}$ because they are obtained by applying $\alpha_{AS}$ on $\tau$. Therefore, $\tau'' \in C_{AS}[m_{AS}]$.

Theorem 4. (Completeness of $C_{AS}[c_{AS}]$)

Let $c_{AS}$ be an AsmL class. Then

$$\forall \tau \in T(\Sigma). \tau \in C_{AS}[c_{AS}]. \exists AS_{P} \in AS_{Pg}. \exists \rho_0 \in AS_{Env} \times AS_{Store}. \exists o_{AS} \text{ instance of } c_{AS}. \exists \tau' \in T(AS_{Env} \times AS_{Store}). \tau' \in W[\rho_0] \wedge \alpha_{AS}((\tau')) = \{\tau\}$$

Proof. An AsmL program satisfying the previous theorem can be constructed by creating and instance of $c_{AS}$ in the Main function, the initial state corresponds to the state when the class constructor, $AS_{Str}$, was executed. It is always possible to construct both $AS_{P}$ and $\rho_0$. For instance, there exist many other possible constructions involving AsmL methods pre-conditions and post-conditions.

5 Application

The SystemC [11] 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 [7] 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 [12].

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.](image-url)

By providing a formalization of the SystemC and AsmL semantics in fixpoint based on the OO general case given in [10], we proved in [8] that, for every SystemC program, there exists an AsmL program preserving the same properties, w.r.t. an observation function \( \alpha \). 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 [12]) 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 checking of AsmL programs.

References