Formal Semantics for PSL Modeling Layer and Application to the Verification of Transactional Models

Luca FERRO, Laurence PIERRE
TIMA (CNRS-Grenoble-INP-UJF)
46 Av. Félix Viallet - 38031 Grenoble cedex - France
Email: Luca.Ferro@imag.fr, Laurence.Pierre@imag.fr

Abstract—The IEEE standard PSL is now a commonly accepted specification language for the Assertion-Based Verification (ABV) of complex systems. In addition to its Boolean and Temporal layers, it is syntactically extended with the Modeling layer that borrows the syntax of the HDL is which the PSL assertions are included, to manage auxiliary variables. In this paper we propose a formal, operational, semantics of PSL enriched with the Modeling layer. Moreover we describe the implementation of this notion in our tool for the dynamic ABV of SystemC TLM models. Illustrative examples are presented.

I. INTRODUCTION

Assertion-Based Verification (ABV) aims at guaranteeing that designs obey properties, usually expressed under the form of logico-temporal formulas (assertions) that capture the design intent [1], [2]. ABV allows to take advantage of advanced verification techniques and to benefit from a significant reduction in simulation debugging time [3]. The assertions can be checked using static (model-checking) or dynamic (simulation-based) techniques. Languages like the IEEE standards SVA [4] and PSL (Property Specification Language) [5] are typically used to formalize them.

The Boolean layer of PSL enables to build basic expressions commonly used by the other layers. The core of the language is the Temporal layer, that gives the possibility to describe complex temporal relations. The Modeling layer is defined to augment what is possible using PSL alone, in particular it allows to manage auxiliary variables (not present in the design) to express more elaborate properties. According to the language reference manual, the Boolean and Modeling layers borrow the syntax of the HDL is which the PSL assertions are included (for instance, we talk about VHDL or Verilog “flavors”), but no formal semantics is given for the Modeling layer. However this feature can be very useful in practice (e.g., for the expression of properties at the transactional level) and needs to be more clearly formalized.

TLM (Transaction Level Modeling) is distinctive of the SystemC language [6], which is in fact a library of C++ classes for modeling circuits. TLM is much more abstract than RTL (Register Transfer Level), it provides communication models for complex data types between the components of the System on Chip (SoC). TLM is gaining acceptance, in particular because the simulation of TLM models is several orders of magnitude faster, thus considerably improving productivity in SoC design [7]. Therefore TLM specifications tend to become golden reference models [8] and must be reliable. In [9] we have presented a methodology that enables the dynamic verification of temporal properties for TLM specifications; we check the validity of PSL assertions that express properties on communications i.e., properties associated with transactional operations. A prototype tool, called ISIS, and experimental results are featured in [10].

The contribution of this paper is twofold. First we propose a formal, operational, semantics of PSL endowed with the Modeling layer. Then we refine it to fit the transactional level, and we describe the implementation of this notion in our ISIS tool. We depict its usefulness on some illustrative examples.

Dealing with additional variables is also addressed for instance in [11], [12], but they mainly concentrate on the concept of local variables and the issues related to their semantics in SVA. This context of SVA local variables is considered at the RT level in [11], and the purpose is to generate hardware components for static verification. The use of local variables is also addressed in [12]. The authors discuss the notion of scope of a variable to overcome some drawbacks of the semantics of local variables in SVA, like the fact that the union and intersection operators do not completely conform to the notions of set union and intersection.

In this paper we focus on the notion of global variables. Since PSL [5] has not been given semantics for auxiliary variables yet (neither global nor local), considering global variables (which is moreover the sole notion supported by the Modeling layer) is a natural starting point.

II. FORMAL SEMANTICS

A. Overview of PSL

The semantics of PSL properties [5] is defined with respect to finite or infinite execution traces, or words over the alphabet \( \Sigma = 2^P \cup \{\top, \bot\} \) (\( P \) is a non-empty set of atomic propositions, \( \top \) and \( \bot \) are such that, for every boolean expression \( b \), \( \top \models b \) and \( \bot \not\models b \)). The length of a trace \( v \) is denoted \(|v|\), the \( i^{th} \) evaluation point over \( v \) is denoted \( v^{i-1} \), and \( v^i \) indicates the suffix of \( v \) starting at \( v^i \). In this paper, we focus on properties of the Foundation Language (FL) class of PSL,
which essentially represents the linear temporal logic. In the following, \( \varphi \) and \( \psi \) are FL formulas.

Here is a short overview of the trace semantics as given in the language reference manual (see [5] for more details):

- **Boolean expressions**
  \[
  v \models b \iff |v| = 0 \text{ or } v^0 \models b
  \]
  where \( \models \) is the satisfaction relation for boolean expressions.

- **Logical connectives**
  \[
  v \models \neg \varphi \iff \bar{v} \not\models \varphi
  
  v \models \varphi \land \psi \iff v \models \varphi \text{ and } v \models \psi
  \]

- **Next operator:** the meaning of the next! operator is that the sub-property denoted by its operand should be verified from the next evaluation point of the execution trace (the weak version next is similar, except that the existence of a next evaluation point is not mandatory):
  \[
  v \models \text{ next!} \varphi \iff |v| > 1 \text{ and } v^{\ell-1} \models \varphi
  \]

- **Strong Until:** until! or \( U \) (classically, the weak until operator until does not impose the occurrence of \( \psi \)):
  \[
  v \models [\varphi U \psi] \iff \exists k < |v| \text{ s.t. } v^{k+1} \models \psi \text{ and } \forall j < k, v^j \models \varphi
  \]

As usual, the PSL formula always \( \varphi \) means that \( \varphi \) must be verified on each evaluation point of the trace. The strong next\_event! operator requires the satisfaction of \( \varphi \) the next time a boolean expression \( b \) is verified:
\[
\text{next\_event!(b)(\varphi) def}= \lnot b U b \land \varphi
\]

**B. Formal Semantics with the Modeling Layer**

The Modeling layer allows to declare and give behavior to auxiliary signals and variables. For the Verilog and VHDL flavors, it consists of the synthesizable subset of these languages, whereas the SystemC flavor consists of those declarations and statements which would be legal in the context of the SystemC module to which the PSL \( \text{vunit} \) is bound [13]. Due to the lack of formal framework, this layer is still rarely used and few examples are available. Here is a very simple example given in the Verilog flavor: an additional signal \( \text{valid\_read\_request} \) is declared and receives the value of an expression evaluated using signals of the design [13].

```
\text{vunit modeling\_example} \{
   \text{wire valid\_read\_request;}
   \text{assign valid\_read\_request =}
   \text{read \&\& read\_en \&\& \{busy;}
   \text{assert always \{valid\_read\_request \rightarrow}
   \text{eventually! data\_valid\};}
\}
```

In this example, the additional signal is used for the sake of readability but is not actually mandatory. The Modeling layer can also be used to manage indispensable new variables (e.g., counters that sum values). Significant examples will be given in section IV.

The reference manual does not give details about the semantics of Modeling layer constructs. It is commonly admitted that the statements of the Modeling layer should be evaluated at each step of the evaluation of the property. To give a formal semantics to an assertion \( \varphi \) augmented with a block of statements \( m \) in the Modeling layer, we have to formalize that the assignments of \( m \) perform side effects on variables that can be used in the expression of \( \varphi \). To that goal, we adopt an operational semantics inspired from the one of [14], instead of a trace semantics like the one given in section II-A. As in [14], we consider that the negation operator only appears at the boolean level, which is coherent with the simple subset of PSL used in the context of dynamic verification.

In the following operational rules, \( \varphi, \varphi', \psi \) and \( \psi' \) denote FL formulas, \( b \) is a boolean expression, \( DV \) denotes the set of all the variables declared in the Modeling layer, \( m \) is the block of statements of the Modeling layer, and \( \rho \) is the current environment (association of variables of \( DV \) with their current values). \( [\varphi]^m_\rho \) is the interpretation of \( \varphi \) in which the variables of \( DV \) are substituted by their values in \( \rho \). Like in [14], a derivation \( \frac{\varphi}{\psi} \) is used: \( [\varphi]^m_\rho \frac{\ell}{\psi} [\psi]^m_\rho \) means that, in order to check if a word starting with the letter \( \ell \) satisfies \( [\varphi]^m_\rho \), one can check that \( [\psi]^m_\rho \) is satisfied by the word without \( \ell \).

- **Logical and (similarly for the logical or):**
  \[
  \frac{\varphi}{\varphi} \frac{\ell}{\varphi} [\varphi]^m_\rho \frac{\psi}{\psi} \frac{\ell}{\psi} [\psi]^m_\rho \frac{\ell}{[\varphi \land \psi]^m_\rho}
  \]

- **Next operators:**
  \[
  \frac{\varphi}{X! \varphi} [\varphi]^m_\rho \frac{\ell}{[\varphi]^m_\rho} \quad \frac{\varphi}{X \varphi} [\varphi]^m_\rho \frac{\ell}{[\varphi]^m_\rho}
  \]

- **Until operators:**
  \[
  \frac{\varphi}{[\varphi U \psi]^m_\rho} \frac{\ell}{[\varphi \land (\varphi U \psi)]} [\psi]^m_\rho \frac{\ell}{[\psi]^m_\rho}
  \]

- **Boolean expressions:**
  \[
  [\ell]^m_\rho \frac{\ell}{T} \quad [\ell]^m_\rho \frac{\ell}{F} \quad \text{if } \ell \models \rho \ b \quad \text{otherwise}
  \]

where \( \ell \models \rho \ b \iff \ell \models \rho (b_{\text{DV} \rightarrow \rho(DV)}) \) is \( b \) in which every identifier of \( DV \) has been substituted by its value in \( \rho \).

In these rules, \( \rho' = [m]^\ell_\rho \) which denotes the environment obtained after executing the statement block \( m \) in the environment \( \rho \) and in the context of the atomic propositions of \( \ell \).

In accordance with [14], iteratively applying these rules on a formula \( \phi \) over the letters of a word \( w \), in an environment
\( \rho \), is written \( \phi(\ell w)_{\rho} \) where
\[
\phi(\ell)_{\rho} = \phi \\
\phi(\ell w)_{\rho} = \phi'(w)_{\rho'} \text{ where } \phi'_{\rho'} \iff \phi_{\rho}^{\rho'}
\]

Then the satisfaction relation, with the initial environment \( \rho_0 \), is defined as follows:
\[
w \vdash \phi \iff ok(\phi(\ell w)_{\rho_0})
\]
where \( ok \) calculates whether a given formula has not been contradicted yet w.r.t. the sequence of letters that has already been visited. We use the same function \( ok \) and extend it similarly for weak and strong versions of the operators i.e.,
\[
ok(F) = false \\
ok(T) = true \\
ok(b) = true \\
ok(\varphi \land \psi) = ok(\varphi) \land ok(\psi) \\
ok(\varphi \lor \psi) = ok(\varphi) \lor ok(\psi) \\
ok(X \varphi) = true \\
ok(X! \varphi) = true \\
ok(\varphi W \psi) = ok(\varphi) \lor ok(\psi) \\
ok(\varphi U \psi) = ok(\varphi) \lor ok(\psi)
\]

Since the work of [14] only considers weak operators, the notion of satisfaction above is sufficient.

However PSL defines four levels of satisfaction for the dynamic verification context (where traces are finite): these levels are relevant as soon as strong operators are involved. \( Holds \) and \( Holds \text{ strongly} \) are two satisfaction relations (for simple \( Holds \), the property may not hold on any given extension of the trace). Both of them required that “No bad states have been seen” and “All future obligations have been met”. The \( Pending \) level is defined as “No bad states have been seen” and “Future obligations have not been met”. Finally, \( Fails \) means “A bad state has been seen”. The \( ok \) function of [14] corresponds to the goal “No bad states have been seen”. Here we define a new function \( met \) that expresses the second goal “All future obligations have been met”. Combining the two functions enables to define the \( Holds \), \( Pending \) and \( Fails \) levels over the letters of a word \( w \):

\[
\text{Holds}_{w, \rho_0}(\phi) = ok(\phi(\ell w)_{\rho_0}) \land met(\phi(\ell w)_{\rho_0}) \\
\text{Pending}_{w, \rho_0}(\phi) = ok(\phi(\ell w)_{\rho_0}) \land \neg met(\phi(\ell w)_{\rho_0}) \\
\text{Fails}_{w, \rho_0}(\phi) = \neg ok(\phi(\ell w)_{\rho_0})
\]

The \( met \) function is defined as follows:

\[
met(F) = true \\
met(T) = true \\
met(b) = true \\
met(\varphi \land \psi) = met(\varphi) \land met(\psi) \\
met(\varphi \lor \psi) = (ok(\varphi) \land met(\psi)) \lor (ok(\psi) \land met(\varphi)) \lor met(\varphi) \land met(\psi) \\
met(X \varphi) = true \\
met(X! \varphi) = false \\
met(\varphi W \psi) = true \\
met(\varphi U \psi) = met(\varphi) \land ok(\psi) \land met(\psi)
\]

III. MODELING LAYER AND THE VERIFICATION OF TLM MODELS

A. Overview of the Method and Tool

A method for the automatic construction of SystemC checkers (monitors) from PSL assertions, and for the observation of communication events, has been specified and implemented in a prototype tool with a graphical user interface [9], [10]. The monitors are linked to the design under test through the observation mechanism, and it remains to run the SystemC simulator on this combination of modules. Any property violation during simulation is reported by the monitors.

The tool now supports the declarations and statements of the Modeling layer, on the basis of the semantics described in section II-B, slightly refined as explained in section III-C. Figure 1 gives a screenshot of the ISIS window that allows to capture the assertion. If declarations and statements of the Modeling layer are present, they are interpreted according to the semantics: since the observation mechanism will trigger an activation of the monitors at each step of the evaluation of their assertions, we automatically insert the Modeling layer statements at the beginning of the monitoring functions (to evaluate the expressions in the appropriate environment). Our monitors for strong operators are also equipped with a “pending” output [9] that encodes the notions discussed in section II-B.

The approaches proposed in [15], [16] and [17] concentrate on model checking SystemC designs. Abstraction techniques are required to get tractable models, or only limited (pieces of) designs can be processed. Simulation-oriented solutions are given in [18], [19] and [20]. With regard to these related works, our methodology has the following main attractive features:

- the statements of the assertions can involve several channels, and this group of channels can be heterogeneous (signals and TLM channels),
- it is relatively few intrusive in the SystemC code, which is augmented with mechanically generated new classes, and undergoes few modifications in its declarative parts,
- it provides a high level of automation: the monitors are automatically generated, and the code is mechanically instrumented with user-guidance through a GUI,
- it is highly efficient, both for the construction of the checkers and during instrumented simulation.

Moreover, to our knowledge, none of the above-mentioned related approaches offer solutions to support the Modeling layer. Some commercial tools also provide for introducing PSL assertions in SystemC designs. The tool Cadence Incisive Unified Simulator supports TLM, but only signals can be involved in the assertions. Other tools like Synopsys VCS only accept RTL descriptions.

B. Boolean Layer - Conditions on the Communications

At the synchronous RT level, the observations that constitute the traces are made on the clock edges but here, observation points are related to the communication actions [9].
According to the PSL reference manual, the “SystemC flavor” of the Boolean layer should allow the use of any SystemC boolean expressions. Thus, atomic propositions are C++ boolean expressions. We give the user the possibility to use these atomic propositions to express conditions on the communications and, through the GUI, we also give him the possibility to select the relevant communications for each property (see on the left side of Figure 1). Being able to identify the critical communication events in a given context is crucial. For instance, depending on his confidence in a channel, the user may prefer to activate the verifications when this channel is written or when it is read.

From the set of selected communication operations, the tool automatically generates ad hoc boolean functions, e.g., write_CALL() here for each selected channel/port, to check the occurrence of write actions. Hence \( \ell.\text{write\_CALL}() \), where \( \ell \) is an instance of one of those selected elements, can then be used in assertions to express the occurrence of a write action.

More generally, \( \text{name.\text{fctname\_CALL}()} \) denotes that the communication function \( \text{fctname} \) of the element \( \text{name} \) has just been called. We also use \( \text{name.\text{fctname\_CALL}.p#} \) to denote the parameter in position \# of function \( \text{fctname} \) (0 is used for the return value).

### C. Modeling Layer and TLM

At the transactional level, observations points are related to the communication actions, and letters of the execution traces are more complex than at the RT level. In particular, they can involve the parameters of these communications. To take that aspect into consideration, we slightly modify the notion of letter: we consider that letters are no more made of atomic propositions, but rather made of pairs \( (id, value) \) where \( id \) can be the identifier of a variable of the design or an identifier like \( \text{name.fctname\_CALL}.p# \) (see section III-B), and \( value \) is the corresponding current value.

Moreover, such a value may be undefined at a given observation point if the corresponding communication action does not occur at that point. To take this aspect into account, we refine the previous semantics with the notion of undefined value (denoted \( \vartheta \) below). The operational rule for boolean expressions has to be modified accordingly. It becomes:

\[
\llbracket b \rrbracket_\rho^\ell \rightarrow \begin{cases} 
T & \text{if } \ell \models_{\rho'} b \\
F & \text{otherwise}
\end{cases}
\]

where \( \rho' = \llbracket m \rrbracket_\rho^\ell \) and

\[
\ell \models_{\rho} b \iff \begin{cases} 
\text{false} & \text{if } \exists \text{ a pair } (id, \vartheta) \in \ell|I_b \\
\ell \models_{\rho} b' & \text{otherwise}
\end{cases}
\]

\( I_b \) is the set of identifiers present in the boolean expression \( b \), and \( b' = b[DV \leftarrow \rho(DV)][I_b \leftarrow \ell(I_b)] \) denotes \( b \) in which all the identifiers of \( DV \) have been substituted by their value in \( \rho \) and all the identifiers of \( I_b \) have been substituted by their value in \( \ell \).

With regard to atomic propositions \( q \), the original PSL semantics states that \( \ell \models q \) iff \( q \in \ell \) [5]. It now becomes \( \ell \models q \) iff \( q[DV \leftarrow \rho(DV)][I_q \leftarrow \ell(I_q)] = \text{true} \).

### IV. APPLICATIONS AT THE TRANSACTIONAL LEVEL

#### A. DMA Example

Let us consider a simple DMA example delivered with the first draft of the TLM 2.0 library and pictured in Figure 2. A master programs the DMA through a memory-mapped router (which is a TLM communication channel) in order to perform transfers between two memories.
Fig. 2. DMA Example

A property P1 of interest is: any time a source address is transferred to the DMA, a read access in the first memory eventually occurs and the right address is used. The expression of this property requires the memorization of the address transferred to the DMA (which is the second parameter of the write operation), to be able to check that this address is actually used when the memory is read.

To avoid altering the original description, the Modeling layer is mandatory. In the property below, req_src_addr is the variable that memorizes this value each time the source register of the DMA is overwritten (we recall that initiator_port.write_CALL.p2 denotes the second parameter of function write):

```haskell
// ---- Modeling layer ----
//HDL_DECLs:
int req_src_addr;
//HDL_STMTs:
if (initiator_port.write_CALL() &&
    initiator_port.write_CALL.p1 == dma_src_reg)
    req_src_addr = initiator_port.write_CALL.p2;
// ---- Assertion ----
//PROPERTY P1:
assert ALWAYS ((initiator_port.write_CALL() &&
    initiator_port.write_CALL.p1 == dma_src_reg)
    -> NEXT_EVENT!(mem1.read_CALL())
    (mem1.read_CALL.p1 == req_src_addr));
```

Test coverage analysis can also easily be performed by the introduction of counters in the Modeling layer, for instance here to evaluate the repartition of read/write operations in different sectors of the memories.

For this example, we ran various simulations in which 200000 memory transfers are performed, data of length 64 bytes are transferred (each atomic transfer carries 4 bytes i.e., 16 transfers are needed). For instance, the monitor for P1 is evaluated 4.2 million times (21 times for each transfer: 5 writings to the DMA + 16 read operations). In that case, the CPU time for simulation without monitoring is 4.97 seconds\(^1\), and 5.54 seconds for the same simulation with the verification of property P1. It takes 6.14 seconds if we both perform the verification of P1 and check coverage.

\(^1\)On an Intel Core2 Duo (3 GHz) under Debian Linux, 2 GB memory

B. Protocol over Faulty Channel

This case study is a SystemC version of the protocol over faulty channel described in [21]. A sender sends messages to a receiver through a channel that can loose or duplicate messages. The receiver sends back acknowledgements through a simple signal ackr; each acknowledgement is the number of the last received message. Depending on the value of the acknowledgment signal, messages are re-emitted or not by the sender. According to [21], two properties are verified:

- we always have \( ackr \leq nb\_sent \leq ackr+1 \), where \( ackr \) is the acknowledge sent by the receiver and \( nb\_sent \) is the number of the last message sent by the sender (P2),
- the acknowledgement sent by the receiver is always equal to the length of the actual data (number of different messages) it received (P3).

Both of them require the Modeling layer. These properties are stated as follows:

```haskell
//HDL_DECLs:
int nb_sent = 0;
//HDL_STMTs:
if (ch.write_CALL())
    nb_sent = (ch.write_CALL.p1).number;
//PROPERTY P2:
assert ALWAYS (ack.read_CALL() -> (ack.read_CALL.p0 <= nb_sent &&
    nb_sent <= (ack.read_CALL.p0+1)));

//HDL_DECLs:
int nmsg = 0;
//HDL_STMTs:
if (ch.read_CALL()) n = (ch.read_CALL.p1).number;
if (n != nmsg) nmsg++; // new message arrived
//PROPERTY P3:
assert ALWAYS (ack.read_CALL() -> (ack.read_CALL.p0 == nmsg));
```

In the simulations we realized, the sender sends 4000 times a random number of messages (between 1 and 4000). The CPU time for simulation without monitoring is 8.97 seconds; 11.25 seconds are needed for the same simulation with the verification of property P2, and 10.50 seconds with the verification of property P3 (only 12.21 seconds are needed for the simulation with the verification of both properties together, because the observation time is “factorized”).

C. Motion-JPEG Platform

Let us finish with a more elaborate example, a Motion-JPEG decoding platform described at the Transaction Accurate level [22], see Figure 3. It embeds a configurable number of processors (here we use one processor), a global memory, a hardware semaphore RAM, and hardware terminals (TTY). The traffic generator takes its data from a MJPEG file, and RAMDAC is the viewer.

Among the properties that can be verified to check the reliability of the communication channel, the following one also needs the Modeling layer: the data that are written on the RAMDAC are exactly the ones that have been transmitted...
by the EU (P4). The expression of this property requires the memorization of the data that are transmitted by the EU, to be compared to the data written on the RAMDAC:

```c
//HDL_DECLS:
unsigned int req_data;
unsigned int written_data;
//HDL_STMTs:
if (eu.write_CALL()) req_data = eu.write_CALL.p2;
if (rdac.write_CALL())
    written_data = rdac.write_CALL.p2;
//PROPERTY P4:
assert ALWAYS (eu.write_CALL() ->
        NEXT_EVENT(rdac.write_CALL())
        (req_data==written_data));
```

In this property, \( \text{req\_data} \) is the variable that memorizes the transmitted data; it is updated when \( \text{eu} \) executes its method \( \text{write} \). For the sake of clarity, we also use the Modeling layer to manage a variable \( \text{written\_data} \) that stores the data written on the RAMDAC. It is not mandatory because its value is checked at the same time it is known (a call to the method \( \text{write} \) of the RAMDAC occurs), therefore \( \text{rdac.write\_CALL.p2} \) could be used directly in the \( \text{NEXT\_EVENT} \) expression.

Simulations that correspond to 10 s of SystemC time require about 18.67 s of CPU time without monitoring, and 21.64 s with the verification of property P4.

Finally it is worth mentioning that, for all the examples, simulations have been performed with and without introducing bugs in the design. All the situations that lead to a violation of the properties have been detected, the monitors are correct by construction. Using the solution developed in [23], we can also produce test programs with a variety of nominal and corner cases.

V. CONCLUSION

The use of auxiliary variables using the Modeling layer of PSL is a crucial issue for the assertion-based verification of TLM specifications. To make it possible, we have both given a formal semantics for PSL augmented with the Modeling layer, and implemented an associated solution in our runtime verification tool. The formal semantics goes beyond the transactional level and can be used in any context.

This work is a first step towards improving our assertion-based verification technique at the transactional level. Supporting the notion of global variables is not sufficient in general. For example, with the Motion-JPEG decoding platform, memorizing the written data into \( \text{req\_data} \) each time new data are transmitted by the EU is sufficient here because the platform does not work for instance in a pipelined way. We are working on a solution for considering simultaneously several evaluations of the assertion when several data are processed concurrently (using different, local, variables).

REFERENCES


1This work is partly supported by the French projects SFINCS (ANR) and SoCKET (FCE)