Improving PSS Test Generation Using Model Checking and Conformance Testing

Philippe Ledent, Radu Mateescu, Wendelin Serwe

Improving PSS Test Generation Using Model Checking and Conformance Testing

Philippe Ledent Radu Mateescu Wendelin Serwe
Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP†, LIG, 38000 Grenoble, France
philippe.ledent@inria.fr radu.mateescu@inria.fr wendelin.serwe@inria.fr

***Abstract*—SoC architectures are complex and notoriously hard to verify. Current industrial practice is still mainly based on testing, with the recent standard PSS easing the generation of system-level tests. In this paper, we suggest to formally express the behavior of a PSS model as a composition of communicating labeled transition systems. This improves the PSS methodology in two ways. First, it becomes possible to formally verify temporal logic properties of the model and increase the confidence in the model and the generated tests. Second, conformance testing techniques improve the coverage of the generated tests.**

***Index Terms*—Formal methods, System-on-Chip, Verification**

I. Introduction

SoC (*System-on-Chip*) architectures integrate a complete computing system on a single chip. SoCs are widely used, for instance, in smartphones or as microcontrollers in objects of the Internet-of-Things. The complexity of SoCs, together with their potential use in critical applications, call for thorough verification; current industrial practice of which is still mainly based on hardware simulators using directed tests and/or execution-time assertions [17].

A major challenge in SoC verification is to devise meaningful system-level tests involving many components of the SoC. Indeed, testing a system-level feature (i.e., the good overall behavior of the SoC) requires to specify the interaction of several components and check the entire information flow through all involved components. Due to the sheer number of possible scenarios, it is difficult, even for experienced verification engineers, to gain confidence in the test coverage [7].

PSS (the *Portable test and Stimulus Standard*) [16] defined by the Accellera consortium[1] of hardware design tool vendors and users, provides a language to specify both a *verification intent* (e.g., a sequence of *actions*) and an implicit description of the SoC's behavior (as a collection of ordering constraints on the actions). PSS also defines a methodology to derive from a verification intent, by automatic inference of (intermediate) actions, complete tests that can be run on the DUT (*Design Under Test*, the actual SoC or a simulator). Hence, PSS is an industrial step towards model-based test generation for SoCs, bringing the combined advantages of automatic exploration of test variants (by inferring different intermediate actions) and platform independence (by separating the computation of the ordering of actions from their concrete implementation).

However, PSS users face several challenges. First, PSS emphasizes the verification intent and limits the modeling of the SoC's behavior to a set of constraints, which makes the actually modeled behavior difficult to grasp. Modeling errors often show up only as unexpected tests yielding false positives (pointing to errors in the test, not in the SoC) and limiting confidence in the model and test coverage. Second, PSS promotes the derivation of complete tests using a backward exploration of the verification intent that favors shortest-path solutions, possibly missing interesting and more complex tests.

The contributions of this paper address these two aspects by formally expressing the semantics of the PSS behavior constraints as a composition of LTSs (*Labeled Transition Systems*). Besides pinpointing ambiguities and semantic options, this paves the way to the application of existing formal methods and tools. First, a model checker increases confidence in the behavior model by verifying interesting properties before generating tests. Second, a conformance testing tool enables the generation of test suites with coverage guarantees. We instantiate our approach using the CADP verification toolbox [3][2] (that received a test of time tool award[3] in 2023) and its companion TESTOR tool [12], [13]. We use ARM's AMBA APB protocol [1] to illustrate the benefits of our approach. We reveal PSS modeling limitations due to semantic ambiguities on a carefully handcrafted small example. Our approach has been applied to more complex protocols [11].

PSS is a rather recent industrial proposal—the latest

†Institute of Engineering Univ. Grenoble Alpes
[1]https://accellera.org/

[2]https://cadp.inria.fr/
[3]https://etaps.org/awards/test-of-time-tool/

version has been released in 2023—so there are still few academic research papers about improving PSS. Close to our work, CoVerPlan [2] proposes specific annotations in PSS to improve the test generation upon the shortest-path for longer, more complex, but still interesting tests using (bounded) SAT solvers, whereas our approach accepts PSS *as is* and uses model checking and conformance testing. Our approach shares with CoVerPlan the generation of tests as completely instantiated PSS code, taking advantage of the portability of PSS and available industrial tools for actually running the tests. MetaPSS [15] suggests to apply meta-modeling techniques inspired from software engineering to ease the generation of PSS models; this is complementary to our approach, which helps understanding PSS behavior models and improve their use. Another complementary work [9] proposed to incorporate the PSS methodology in an existing industrial verification workflow by mapping PSS models onto the UVM (*Universal Verification Methodology*) environment.

The remainder of this paper is organized as follows. Section II presents PSS and its use to generate tests for ARM's APB protocol. Section III shows how to express PSS modeling constructs as compositions of LTSs and discusses ambiguities and semantic options of PSS on a dedicated example. Section IV illustrates the benefits of model checking for the early detection of modeling errors prior to test generation. Section V improves PSS test generation using conformance testing tools. Finally, Section VI concludes.

II. Testing the APB Protocol with PSS

This section illustrates test case generation for ARM's APB (AMBA Peripheral Bus) protocol [1] using PSS, which involves three steps: expressing the verification intent, modeling the behavior, and the actual generation of test cases.

A. APB (AMBA Peripheral Bus) Protocol

The APB protocol is a low-cost, low-power, and low-speed SoC bus protocol for accessing peripheral registers. Transactions of the protocol are between an initiating *requester* and a responding *completer*. Requesters are usually in other high-speed parts of the SoC communicating with other bus protocols (AXI, AHB). The connection between the protocols is organized by a *bridge* (AXI2APB, AHB2APB) which acts as the requester in APB transactions. APB is simple, but sufficiently realistic to illustrate our work (and also [2]).

Fig. 1 shows the state machine for APB communication. The states depend on the value of the signals. The labels of the transitions give interpretations to signal changes. The initial action INIT_STATE usually sets all signals to 0 and in particular pselx indicating that at first

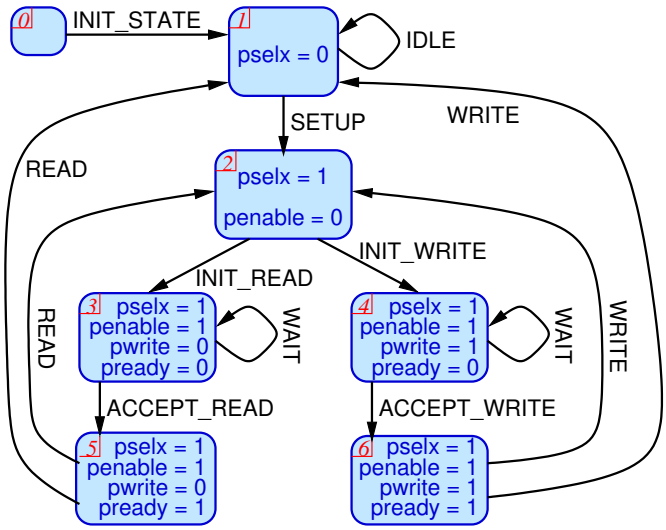


Fig. 1: APB protocol state diagram

no peripheral is selected. When pselx becomes 1 with the transition SETUP, the completer peripheral identified by x is chosen. To perform a *read* or a *write*, the requester first sets the pwrite signal, the address bus, and the data bus (when appropriate, i.e., when writing). The requester then sets the penable signal to 1 to commit to the transaction, and possibly WAITs until the completer indicates it is ready to ACCEPT the transaction by setting pready to 1. Upon completion, the protocol either goes back to state 2 (SETUP) if another transfer is immediately possible, or back to state 1 (IDLE) otherwise. Figure 1 features nondeterminism for transitions READ in state 5 and WRITE in state 6 to reflect the nondeterminism of the APB protocol [1]. In practice, the protocol is deterministic depending on how the requester drives the pselx signal.

B. Expressing Verification Intent in PSS

Atomic actions are the main building blocks of PSS and abstract the concrete behavior (expressed in another language) a test should execute on the DUT. In the sequel and whenever there is no risk of confusion, we use the term "action" to denote an "atomic action".

Tests are generated for a *verification intent*, expressed as a *compound action* i.e., a composition of atomic actions using operators borrowed from concurrency theory, e.g., sequential composition, non-deterministic choice, and parallel composition. The verification intent is contained in the top component of the PSS model (usually called pss_top).

One verification intent for the APB is to require the execution of a READ (lines 51–53 of Fig. 2). Note that a READ cannot be executed immediately, and the PSS methodology completes the test by inferring sequences

```
component pss_top {
 // types
enum bit32 {32b0, 32b1}; // data
enum bit8  { 8b0,  8b1}; // address
enum bit1  {  b0,   b1}; // signal
 // flow objects
state signals {
  rand bit1  pselx;
  rand bit1  penable;
  rand bit1  pwrite;
  rand bit1  pready;
  rand bit8  address;
  rand bit32 bus_data;
  rand bit32 slave_data_0;
  rand bit32 slave_data_1;
};
pool signals apb_signals;
bind apb_signals *;
 // action read
action READ { ... };
 // action write
action WRITE {
  input  signals apb_in;
  output signals apb_out;
   // input constraints (ACCESS(6) state, ready, write)
  constraint apb_in.initial == false;
  constraint apb_in.pselx   == b1;
  constraint apb_in.penable == b1;
  constraint apb_in.pwrite  == b1;
  constraint apb_in.pready  == b1;
   // output constraints (SETUP(2) or IDLE (1))
  constraint apb_out.penable == b0;
   // no constraint on apb_out.pselx to choose between SETUP(2) and
IDLE (1)
   // constraints to reduce the state space by resetting unused fields
  constraint apb_out.pready    == b0;
  constraint apb_out.pwrite    == b0;
  constraint apb_out.address   == 8b0;
  constraint apb_out.bus_data == 32b0;
   // effect on the memory
  constraint (
    ((apb_in.address == 8b0) &&
      (apb_out.slave_data_0 == apb_in.bus_data) &&
      (apb_out.slave_data_1 == apb_in.slave_data_1)) ||
    ((apb_in.address == 8b1) &&
      (apb_out.slave_data_0 == apb_in.slave_data_0) &&
      (apb_out.slave_data_1 == apb_in.bus_data)));
};
 // Other actions (for all transitions of Figure 1)
 ...
 // Verification Intent
action INTENT_READ {
  activity { do READ } // anonymous reference to READ
};
}
```

Fig. 2: Excerpt of the PSS code for the APB (see Fig. 1)

of atomic actions that must be executed before a READ. This requires a behavioral model defining all atomic actions and their ordering constraints.

C. Modeling Behavior in PSS

In PSS, the behavior is modeled implicitly by ordering constraints on actions and their interactions through *flow objects* for synchronization and communication. A flow object may have *fields*, which are initially set to an explicit value or a random one (using the keyword rand).

Each action may read from flow objects, execute, and then may write to flow objects. Note that the sequence of these three steps (read, execute, write) is not atomic in the sense that they might interleave with steps of other actions.

There are three kinds of flow objects: *state*, *stream*, and *buffer*. Before any action can read its inputs from buffer and stream flow objects, these must be written to as another action's output. Actions express constraints on the content of the flow objects they read from and write to. Reading from and writing to a flow object accesses all its fields simultaneously. The authorized ordering of actions is determined by *scheduling constraints*, which can be explicit when actions restrict the domain of definition of data interaction with flow objects, or implicit through the internal behavior of flow objects. The set of allowed values for a flow object's fields is determined by the conjunction of constraints from the objects themselves and the actions they interact with.

State flow objects represent concurrent-read exclusive-write variables shared between actions; each one has an implicit Boolean field initial, which reads true until the first write, after which it always reads false, even if all other fields have their initial value. Writing to a state *replaces* all its fields except initial. From a formal standpoint, a state flow object is a tuple of variables for which the content is used to explicitly enumerate states, ranging from 0 to $2^b - 1$ where b is the maximal number of bits needed to represent the size of the cross-product of the fields.

Stream flow objects allow two actions executing "at the same time" to synchronize and exchange data. This implies that the actions must be running in *parallel*.

Buffer flow objects are persistent write-once concurrent-read memory cells, to which an action writes data that might then be read by other actions. A buffer must be written before being read and subsequently can only be read. Every write corresponds to a creation of a new buffer. [4]

A flow object *pool* groups flow objects of a same kind. Declaring a flow object in PSS just defines a type for such flow objects, which are truly instantiated through pools. The bind statement binds a pool to a set of actions (specified by a component hierarchy name path, possibly ending with the wildcard "*") indicating that only actions from that set might read from or write to flow objects within the pool. These bindings are another level of scheduling constraints.

A pool of states can contain at most one state. A pool of streams can contain as many streams as necessary, based on the actions that need to communicate through

[4] Note the difference to the usual notion of a buffer, where an element is expected to be removed when read.

them. A pool of buffers actually behaves more like a bag: writes *accumulate* values (buffers with new combinations of data on each field); reads obtain *any* value previously written (because reads do not remove buffer flow objects from the pool). Hence a pool of buffers can contain at most as many (different) buffers as there are combinations of data for the buffer's fields, because there is no need to store identical buffers several times. The maximal size of a pool of buffers is finite: it is the maximal number of data combinations a buffer's fields can take, i.e., the cross-product of the domains of each field.

We encode the APB state diagram of Fig. 1 as a state machine with a single state flow object and an action for every transition (arrow) of Fig. 1. An excerpt of the resulting PSS code is shown on Fig. 2. The state flow object (lines 7–16) contains all major signals of the APB protocol needed for the state machine—we only omit the check, clock, protection, reset, strobe, and the user signals [1]. To prevent state-space explosion, we replace 32-bit and 8-bit bit-vectors by enumerated types (lines 3–5).

We illustrate the encoding of a transition by the WRITE transition yielding the WRITE action (lines 22–47). Each action reads from and writes to the state flow object (lines 23–24). The constraints on the input fields (lines 26–30) characterize the source state of the transition, i.e., that pselx, penable, pwrite, pready must be b1. Unconstrained input fields may hold any values. Similarly, the constraints on the output fields characterize the target state of the transition, i.e., penable must be b0 (line 32). There are also constraints to express the effect on the memory: the data stored at the address should take the value of bus_data (lines 41–46). Unconstrained fields of the output flow object are assigned nondeterministically (as for pselx, line 33): it is thus mandatory to indicate the fields that have to be maintained (lines 43 and 45). The remaining constraints (lines 35–38) reset fields that are no longer used to their initial values; this drastically reduces the size of the underlying state space [5].

D. PSS Test Case Generation Methodology

A possible algorithm to generate a test case for a verification intent is given in [16, Appendix E] as a "backward inference", starting with the last atomic action(s) of the verification intent and filling its gaps while solving the scheduling constraints:

1) For every compound action, "flatten" it into a partial ordering of atomic actions.
2) For every atomic action, find all flow objects it can read from that satisfy its constraints.

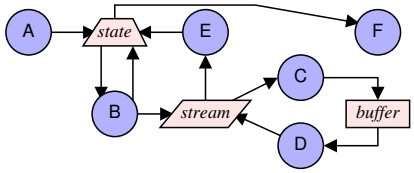


Fig. 3: PSS example with all kinds of flow objects

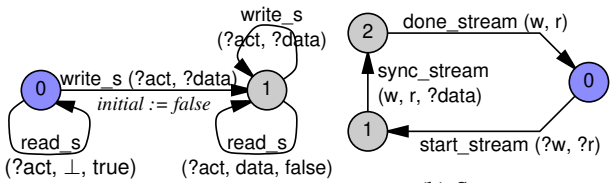
3) For every flow object, select an action that could have written to it, giving preference to actions occurring in the verification intent.
4) Repeat steps 1 to 3 as long as the verification intent is not covered and until there is an action that can start the test, i.e., an action without input flow object or reading only from a state, as state flow objects do not require being written before being read.

For the example of the APB, this algorithm yields a single test case, namely the shortest path to a read, i.e., the sequence "init_state; setup; init_read; accept_read; read". Unfortunately, this test cannot detect that a DUT incorrectly implements a WRITE, simply because the latter is never executed.

III. Expressing PSS Modeling Constructs as Labeled Transition Systems

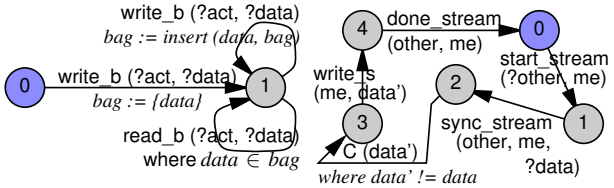
In this section, we propose a formal semantics for PSS as a composition of LTSs (*Labeled Transition Systems*), which are state-transition systems carrying all information in the transition labels. Each action and each flow object is encoded as a separate LTS; their synchronized parallel composition ensures the ordering of the behavior as specified by the PSS semantics. Synchronization between these LTSs is expressed using rendezvous on *gates*, enabling data to be exchanged between the participating LTSs by means of *offers*. We generate *cyclic* LTSs for PSS actions to allow a same action to figure several times in a test in the case of cyclic dependencies (e.g., an action reads from a state flow object and writes to the same state flow object). Note that as a composition of a statically known number of finite LTSs, the resulting LTS is finite.

We illustrate our semantics on a carefully crafted minimal example with six actions (from A to F), passing around one out of two data values data1 and data2 of an enumerated type data_e and ordered using all three kinds of flow objects, as shown on Fig. 3 (the APB example does not use all kinds of flow objects). The state flow object has a Boolean field indicating whether action B is *allowed*. A writes to the state, B reads from the (initialized) state (with allowed) and writes to the state and the stream, C reads from the stream and writes (the other data) to the buffer, D reads from the buffer and



(a) State (b) Stream

Fig. 4: Automata for state and stream flow objects



(a) Pool of buffers (b) Action C

Fig. 5: Automata for a pool of buffers and action C

writes to the stream, E reads from the stream and writes to the state, and F reads from the (initialized) state.

A. LTSs for Atomic Actions and Flow Objects

We define the semantics of actions and flow objects by giving a symbolic automaton for each action and flow object of the example in Fig. 3. The initial state of each automaton is numbered 0. The transitions are labeled with symbolic (rather than concrete) offers for data communication. Offers starting with a question mark "?x" indicate that variable x receives any data of the appropriate type during synchronization (with another automaton). The LTS corresponding to a symbolic automaton is obtained by instantiating all possible values present in the transition labels.

PSS supports the same data types as C/C++. However, generating tests or an LTS requires the exhaustive enumeration of all possible data values. For classical data types, like **int**, this leads to a high number of tests or a very large LTS. Often, data-independence or static analysis allow to drastically reduce the set of different data values, without losing interesting behaviors, and to use small enumerated data types as in Fig. 2. Hence, we focus on varying the *behavior* of the tests (i.e., the ordering of actions to be executed), and leave the instantiation of abstract data values into (randomly chosen) concrete values to the test execution tool.

We first present the LTSs for flow objects, because they impact those for actions. Each flow object has two gates "read_x" and "write_x" with an offer for each field of the flow object (i.e., fields are simultaneously read or written). In the corresponding LTSs, each state of the symbolic automaton is expanded into as many LTS states as there are (combinations of) different values for the fields.

A *state* flow object (see Fig. 4a) is akin to a memory cell holding data plus an implicit Boolean field initial (that cannot be explicitly written). The two states of the automaton reflect the value of initial , which is initialized to true, already enabling reads in the initial state, but with an undefined value for the data. Once written, the initial field will always read false even if all other fields have their initial values.

A *stream* flow object (see Fig. 4b) enables parallel actions to communicate directly, naturally expressed by a three-way rendezvous between the two actions and the stream flow object. Transition sync_stream corresponds to the rendezvous: the first offer (w) identifies the writing action, the second offer (r) identifies the reading action, and the third offer corresponds to the exchanged data. Each involved action has to contain a corresponding rendezvous. An action writing to the stream must provide a concrete value for the first and third offer; the second offer is provided by the reading action. Because an action has no knowledge with which other action it communicates, it is up to the stream to check and enforce that the actions taking part in the rendezvous belong to the set of authorized actions (established by the binding and stored in a local variable of the stream). Two further three-way rendezvous ensure that actions w and r execute at the same time: start_stream marks the beginning and done_stream the end of the synchronization.

The semantics of a *buffer* flow object is persistent data: once data is written, it can be read by any action arbitrarily often. The Fig. 5a represents a *pool* of buffers. It records in the variable bag (initialized to the first written data) the set of all written data and allows to read any data in bag.

In general, the symbolic automaton for an atomic action has three parts: read all inputs, perform the action, write all the outputs. If any of the input or output flow objects happens to be a *Stream*, then the very first transition synchronizes on the gate start_stream, and the last transition on the gate done_stream.

The symbolic automaton in Fig. 5b expresses the semantics of action C. C starts the synchronization with the (input) stream flow object, retrieving other, the identifier of the other synchronizing action. The second transition synchronizes with the stream and receives data. The third transition represents the execution of the action, which (according to its constraints) starts by choosing nondeterministically data' different from data and displays the name of the action and data'.[5] The fourth transition

[5]For more informative transition labels, one could include also the data read from the stream.

```
1  par write_s, read_s, write_b, read_b in
2    par buffer || state end par
3  || par
4       A || F -- actions without streams
5    || par start_stream, sync_stream, done_stream in
6            stream
7       || par start_stream#2, sync_stream#2,
8                done_stream#2 in
9                B || C || D || E -- actions with streams
10           end par
11      end par
12   end par
13 end par
```

Fig. 6: Overall parallel composition for the example of Fig. 3

writes data' to the (output) state flow object. At last, the synchronization with the stream is finished.

The PSS standard is unclear whether *compound actions* should be considered part of the behavior. We choose to consider that compound actions should be used only as verification intents, because constraints induced by compound actions only remove behavior from the LTS generated from the PSS model considering only actions and their interaction constraints with the flow objects. Also, the restrictions induced by a compound action could be expressed easily by appropriate constraints in flow objects (e.g., by additional fields in a state flow object).

B. Overall Behavior as Parallel Composition of LTSs

In a nutshell, the overall behavior corresponds to the parallel composition of two groups of LTSs, synchronizing on shared gates. The first group contains the LTSs of all actions, which interleave without direct interactions. The second group contains the LTSs of all flow objects, which also interleave without direct interactions. By definition of the synchronized parallel composition, the resulting LTS satisfies all scheduling constraints; in case the latter are not satisfiable, the obtained LTS is a single state without any transitions.

In practice, the three-way rendezvous for streams requires a more involved parallel composition, which can be expressed in the EXP language [10] as shown in Fig. 6. The **par** operator is n-ary and allows the composed automata to synchronize on a global set of gates (between the **par** and **in** keywords as in the top-level operator at line 1). The default synchronization paradigm is *maximal cooperation*: all automata composed by a **par** operator must synchronize on g, i.e., none of these automata can execute g alone but has to wait for all other automata to participate in the synchronization on g. Streams are separated from buffer and state flow objects, but grouped (lines 5–11) with all actions connected to a stream. The latter actions are synchronized two-by-two (lines 7–8) using the m-among-n parallel composition operator[6] [6], which refines maximal cooperation.

C. Discussion of Semantic Options

Defining a PSS behavior semantics is challenging due to conflicting interpretations of the natural language and unspecified behavior of actions without input or output flow objects.

Actions communicating by a stream should be executed "*at the same time*" whereas those communicating by a state or buffer are *sequential*. Consequently, two actions mutually communicating through a state and a stream such as B and E in Fig. 3, along with a strict interpretation of "at the same time", introduce a concurrent read/write conflict in the state flow object: the ordering of B and E cannot be both "at the same time" and sequential. Tools with such strict interpretations can never find a test where E immediately follows B and actions C and D will always be inserted in between.

However, a fine analysis of the ordering requirements reveals that actually B must execute *before* E (even in a parallel "at the same time" execution), because B *sends* data over the stream to E which *receives*. For this reason, our implementation of streams ends up serializing B and E.

Furthermore, when an instance B1 of B sends some data to E on the stream, another instance B2 of B must read what E wrote to the state. Therefore, in principle, tools that consider a conflict are not required to do so.

For actions without inputs or outputs, we distinguish four different semantic options, which we classify as *input-enabled*, *input-disabled*, *output-enabled*, and *output-disabled*.

On the one hand, an action with no input flow object has no scheduling constraints about preceding actions. Hence, it can either be considered *input-enabled*, always available (constraints always true) or *input-disabled*, available only at the start of the test run. The PSS methodology implements the latter (because the backward inference stops at actions without inputs), but adopting the former might lead to unusual, yet interesting test cases.

On the other hand, an action with no output flow object has no constraints on succeeding actions. Hence, either anything could happen with *output-enabled*, or nothing at all with *output-disabled*. The PSS methodology implements the latter (because the execution of an action without an output is not necessary to complete the verification intent, the backward inference will never infer such an action if it does not occur explicitly in the verification intent), but including such actions might also lead to unusual, yet interesting test cases.

[6]The #2 requests that exactly two actions participate in the synchronization.

```
1 | [ true* . {ACCEPT_WRITE} . i* . {READ ...} ] false
2 | [ true* . {INIT_WRITE ?a:String ?d:String} ]
3 |   FAIRLY ({WRITE !a !d})
4 | [ true* . {INIT_WRITE ?a:String ?d:String} .
5 |   not {WRITE !a ...}* .
6 |   {WRITE !a ?e:String where e <> d} ] false
```

Fig. 7: MCL formulas for model checking the APB protocol

D. Implementation

We implemented the LTS-based semantics in a prototype translator, named PSS2LNT (about 15,000 lines of code), which parses (correct) PSS code and produces LNT [4] code from which the corresponding LTSs and their composition can be generated using tools of the CADP toolbox [3].

IV. Benefits of Model Checking

Given the implicit nature of a PSS model and the difficulties of an incremental development (because the effect of adding actions is hard to predict), gaining *confidence* in the PSS model is a nontrivial task. In practice, the tests generated from a PSS model are often surprising, due to missing or undesired constraints in the PSS model. The possibility to generate and analyze the underlying LTS clearly increases confidence in the PSS model and test generation. Concretely, the LTS produced by PSS2LNT and the CADP toolbox for the APB protocol is an expansion of the automaton of Fig 1, where each transition is instantiated for all possible values of its offers. For the READ and WRITE actions, there are four possible combinations of address_bus and data_bus (two values for each of these offers).

Hence, checking the reachability of all PSS actions in the LTS (by inspecting its labels) helps detecting modeling errors. With this technique, we actually spotted a mistake (due to an uncareful copy-paste) in our initial version of the PSS code for the APB: we observed that the model contained no WRITE transitions, although ACCEPT_WRITE transitions were present.

To understand and correct the problem, we model checked several temporal logic properties, expressed as the MCL [14] formulas shown on Fig. 7. We first checked a safety property, expressed by a necessity modality $[R]$ **false**, where R is a regular expression denoting the undesirable transition sequences and **false** is a state subformula that forbids them. Thus, the MCL formula (line 1) forbids that the next visible transition after an ACCEPT_WRITE is a READ transition (the wildcard notation "…" denotes a list of any offers). The counterexample provided by the model checker pointed to an incorrect constraint, due to which the PSS action ACCEPT_WRITE resets pwrite, transforming every write into a read.

After correcting this error, we observed that all READ and WRITE transitions accessed address 8b0, although there were also INIT_READ and INIT_WRITE transitions for address 8b1. The second formula (lines 2–3), expressing a response property, requires that each INIT_WRITE is necessarily followed (in a fair manner, to avoid WAIT cycles, see Fig. 1) by a WRITE with the same address a and data d (note the capture of the values of a and d in line 2 and their reuse in line 3). The counterexample pointed to a deadlock after an attempt to write to address 8b1, due to the combined effect of a constraint requiring PSS action WRITE to keep bus_data unchanged and to be reset.

Finally, we used the third formula (lines 4–6) to check the safety property stating that each INIT_WRITE cannot be followed by a WRITE with different offers. The counterexample pointed to a missing constraint: PSS action ACCEPT_WRITE did not constrain the field bus_data to remain unchanged, allowing the written value to be (silently) changed to an arbitrary data.

Without model checking, we might have discovered these bugs in the initial version of our PSS model only through the analysis of tests failing on the DUT. Even worse, in the case of a faulty DUT, bugs in the PSS model and in the DUT could have remained hidden.

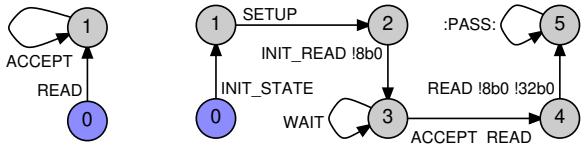
V. Test Generation with Conformance Testing

An alternative to the PSS methodology for test generation is *conformance testing* guided by test purposes [8], which takes as input an LTS (i.e., the PSS behavior) and a TP (*test purpose*, i.e., the verification intent) and produces a CTG (*Complete Test Graph*), a state-transition system which by definition contains all paths of the LTS satisfying the TP: hence, any TC (*test case*) satisfying the TP is by construction included in the CTG. Furthermore, a test suite (set of TCs) covering all transitions of the CTG can be extracted automatically [13]. Then, coverage of all transitions of the LTS can be achieved by a test plan (set of TPs), the CTGs of which cover the LTS; such a test plan can also be computed automatically [13].

We propose to compute tests with conformance test generation tools, e.g., TESTOR [12], taking the verification intent as TP. TESTOR can work on the fly, exploring, as in the PSS methodology, only the required part of the state space.

A. From Verification Intents to Test Purposes

A PSS verification intent is a *compound action* (**activity** keyword), i.e., a wrapper around atomic actions to express additional ordering constraints, using constructions that are easily transposed as compositions



(a) Test purpose (b) Test case 38

Fig. 8: Test purpose and test case for the APB example

```
1  action TEST_CASE_38 { // test 38 out of 40
2    activity { // give an ordering of actions
3      atomic { // no inference
4        do INIT_STATE;
5        do SETUP;
6        do INIT_READ with address == 8b0;
7        do WAIT; // action body must be implemented as a loop
8        do ACCEPT_READ;
9        do READ with (address==8b0) && (data_bus==32b0);
10     }
11   }
12 }
```

Fig. 9: PSS code for test case 38 from Fig. 8b

of LTSs, e.g., conditional branching (**if-then-else**), non-deterministic choice (**select**), sequential composition, or parallel composition.

Verification intent intent_read (lines 51–53 of Fig. 2) is translated into the simple test purpose LTS of Fig. 8a. TESTOR computes the CTG (46 states, 142 transitions, and 22 labels) containing all possible ways to fulfill the TP according to the model. The test case extraction yields a test suite with 40 TCs (with, on average, 28 states and 34 transitions) covering all transitions of the CTG. The TP requests "eventually read any value at any address", thus the CTG contains all possible combinations of actions before the first READ, in particular sequences of WRITE actions preceding the READ.

One TC produced by our methodology is displayed in Fig. 8b. It is the shortest TC that fulfills the TP, reading immediately the initial value 32b0 at address 8b0 without any WRITE action. By this, we show that we also find the shortest test (like the PSS methodology), but obviously this test alone is insufficient to cover all transitions of the CTG—for this reason, TESTOR extracts 39 other TCs.

Fig. 9 illustrates how we translate a TC back into PSS code requiring no inference (**atomic** keyword) and directly executable by industrial tools. Notice that the WAIT self-loop at state 3 of Fig. 8b is serialized at line 7. Actions such as WAIT and ACCEPT_READ actually observe the DUT's behavior by *monitoring* how the *completer* peripheral drives the pready signal. At PSS modeling level it is impossible to know how long pready stays at 0 (transition WAIT) and when it rises to 1 (transition ACCEPT_READ). Thus, we leave the implementation of such a monitoring mechanism to the body of the actions.

verification intent	sem. opt.	CTG states	CTG trans.	LTS coverage	choices	TC number	TC avg. length
intent_BE	DD	8	11	21%	8	4	8.00
	EE	87	412	7%	399	262	14.88
intent_BnoCE	DD	4	3	6%	0	1	3.00
	EE	12	41	1%	32	23	8.13
intent_E	DD	8	11	21%	8	4	8.00
	EE	87	412	7%	399	262	14.88
intent_F	DD	15	52	100%	51	36	8.27
	EE	62	323	6%	322	220	12.39

TABLE I: Test generation statistics

B. Impact of semantic options on coverage

The semantic options from sect. III-C have an impact on test generation and model coverage by test suites. This is best illustrated on the carefully crafted example from Fig. 3. We denote the four semantic options by using two letters: the first for "input", the second for "output", using "E" for *enabled* and "D" for *disabled*. Among these four options (DD, DE, ED, EE), we focus on the extremes DD and EE: for the example in Fig. 3, the (reduced) LTS for DD (resp. EE) has 15 (resp. 1019) states and 52 (resp. 5738) transitions.

We consider four verification intents: intent_BE ("eventually B followed by eventually E"), intent_BnoCE ("eventually B immediately followed by E without C in between"), intent_E ("eventually E"), and intent_F ("eventually F").

Table I shows for each of these verification intents and for semantics DD and EE, the size of the (reduced) CTG (which contains all ways of fulfilling the verification intent in the LTS), the coverage, the number of choices between different tests contained in the CTG, and the number of TCs required to cover all these choices (one TC may cover several choices). All TCs for this example are sequences: their average length is also the average number of states and transitions.

We measure *coverage* of the LTS (PSS behavior) as the percentage of LTS transitions covered by the CTG. This coverage is *maximal* because, by definition, the CTG contains all possible tests for a verification intent [8]. Our approach reaches this coverage by generating a test suite covering all transitions of the CTG [13]. Other approaches may cover less, by giving priority to actions from the verification intent (cutting other meaningful parts of the LTS in Fig. 3) or failing to detect dependency cycles (possibly causing test generation to behave like a *shortest path* exploration of the PSS behavior).

For example, intent_BE requires a test with action B and then eventually E. For DD semantics, Fig. 10 shows its CTG after hiding verdicts and quiescent transitions [8] and minimization. States 2, 5, 6, and 7 each have two

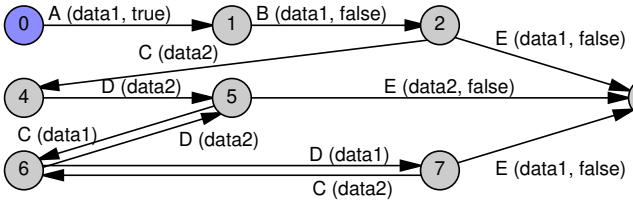


Fig. 10: CTG for verification intent intent_BE with DD semantics

choices that can be covered with four tests (as by our approach), namely the sequences:

1) A; B; E
2) A; B; C(data2); D(data2); C(data1); D(data1); E
3) A; B; C(data2); D(data2); C(data1); D(data2); E
4) A; B; C(data2); D(data2); C(data1); D(data1); C(data2); D(data2); E

It would actually be possible to cover the CTG with only three tests (by omitting test case 3), which suggests improvements of the heuristics in [13]. These tests differ in the point where action E is chosen: immediately after B or after one or two executions of C, as well as the data contained in the state flow object. C reads a data value from the state and writes another value to the buffer, and D reads a nondeterministic value from the buffer. The enumerated type data_e having only two values, the buffer contains all possible values after only two executions of action C, so that any further execution of C does not increase coverage of the CTG.

In contrast, a shortest path approach for a DD semantics only finds the test "A; B; E"[7](3 transitions, 6% coverage) for intent_BE, intent_BnoCE, and intent_E: this is only optimal for intent_BnoCE. Worse, a shortest path approach yields the single test "A; F" (2 transitions, 4% coverage) for intent_F, whereas our approach yields a test suite with 39 TCs and 100% coverage.

Note that the verification intent intent_E (requiring eventually E) yields a CTG equivalent (strongly bisimilar) to the one for intent_BE, because all paths reaching E must pass through B in Fig. 3. Hence the generated TCs are exactly the same.

Semantic options also influence coverage. For action A (without input flow objects), *input-disabled* semantics (DD, DE) force A to occur only once at the beginning of a test, as recommended by PSS. To the contrary, *input-enabled* semantics (ED, EE) allow A to be executed at any time and write to the state, resetting the allowed field to **true**, so that B can re-execute, increasing the size of the CTG and the number of TCs needed to cover it.

[7]Actually, actions communicating via a stream should be executed "at the same time". Thus, the test "A; B; E" should read "A; (B||E)", where "B||E" indicates the simultaneous execution of B and E, e.g., in two threads. Because B *sends* and E *receives*, B necessarily executes *before* E and corresponds to the sequence "B; E".

Now consider action F, which has no output flow objects. *Output-disabled* semantics (DD, ED) inhibit the execution of other actions *after* F. If the verification intent would contain further actions after F, no test could be generated. The "backward inference" methodology of PSS will include F in a test only if it is the last action of the verification intent, promoting the *output-disabled* options (DD, ED). To the contrary, *output-enabled* options (DE, EE) allow further behavior. All actions available before choosing F would remain available after F. Also, F (only reading its input) could now be executed at any point, not influencing the data flow, but constantly creating a choice between executing F or not. This also explains the low coverage for EE: test case generation stops as soon as the test purpose is reached: contrary to DD, F might be followed by many other actions in EE.

All semantic options (DD, DE, ED, EE) are valid, and knowing the option implemented by a tool helps in controlling the generated tests. DD seems suggested by the PSS methodology. EE is the most permissive (and complete), being especially of interest for generating lengthy tests to stress a system. Choosing EE is thus tempting, but causes a drastic coverage drop and requires more verification intents to reach a coverage target (a set of verification intents achieving full coverage could be derived from automatically computed test purposes [13]). The drastic drop in CTG model coverage between DD and EE semantics is explained by the fact that as soon as the verification intent is met, the exploration of the model stops, although the model contains behavior *after* the last action of the verification intent.

VI. Conclusion

Based on a formal semantics for PSS behavior, we suggested two improvements of test generation using PSS. First, formal analysis of the PSS behavior by model checking helps early discovery of modeling errors, increasing confidence in the generated tests and avoiding false positives. Second, using conformance testing, it becomes possible to generate a set of test cases with coverage guarantees, helping to find more bugs in the DUT. Last but not least, formalizing the natural language specification of PSS also pinpoints semantic options and ambiguities. We have implemented prototype tools to generate the LTS corresponding to a (correct) PSS model, generate test cases covering the model, and translated them back into executable PSS tests. These tools have already been applied to test resource isolation for an industrial SoC [11].

As perspective, it seems appropriate to devise a set of generic sanity properties to be systematically checked on the behavior model. We mentioned the reachability of all actions, but there might be others. Similarly, it seems useful to link the verification intents with the properties

checked on the model. On the one hand, verifying that the model satisfies a property derived from a verification intent limits the risk of a false positive during test execution. On the other hand, deriving verification intents from properties yields valuable tests checking for desirable features.

Acknowledgements

Part of this work was supported by a French government grant managed by the *Agence Nationale de la Recherche* under the France 2030 program, reference ANR-23-PECL-0002.

REFERENCES

[1] ARM, "AMBA APB Protocol Specification," arm, ARM IHI 0024E ID022823, Sep. 2023. [Online]. Available: https://developer.arm.com/documentation/ihi0024/latest/

[2] S. Das, S. Sanyal, A. Hazra, and P. Dasgupta, "CoVerPlan: A Comprehensive Verification Planning Framework Leveraging PSS Specifications," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 28, no. 1, Jan. 2023.

[3] H. Garavel, F. Lang, R. Mateescu, and W. Serwe, "CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes," *Springer International Journal on Software Tools for Technology Transfer (STTT)*, vol. 15, no. 2, pp. 89–107, Apr. 2013.

[4] H. Garavel, F. Lang, and W. Serwe, "From LOTOS to LNT," in *ModelEd, TestEd, TrustEd – Essays Dedicated to Ed Brinksma on the Occasion of His 60th Birthday*, ser. Lecture Notes in Computer Science, J.-P. Katoen, R. Langerak, and A. Rensink, Eds., vol. 10500. Springer, Oct. 2017, pp. 3–26.

[5] H. Garavel and W. Serwe, "State Space Reduction for Process Algebra Specifications," *Theoretical Computer Science*, vol. 351, no. 2, pp. 131–145, Feb. 2006.

[6] H. Garavel and M. Sighireanu, "A Graphical Parallel Composition Operator for Process Algebras," in *Proceedings of the IFIP Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification (FORTE/PSTV'99), Beijing, China*, J. Wu, Q. Gao, and S. T. Chanson, Eds. Kluwer Academic Publishers, Oct. 1999, pp. 185–202.

[7] N. B. Harshitha, Y. G. Praveen Kumar, and M. Z. Kurian, "An introduction to universal verification methodology for the digital design of integrated circuits (ic's): A review," in *2021 International Conference on Artificial Intelligence and Smart Systems (ICAIS)*, 2021, pp. 1710–1713.

[8] C. Jard and T. Jéron, "Tgv: Theory, principles and algorithms – a tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems," *Springer International Journal on Software Tools for Technology Transfer (STTT)*, vol. 7, no. 4, pp. 297–315, Aug. 2005.

[9] K. K. Kragseth, "Efficient verification with portable stimulus," Master's thesis, Norwegian University of Science and Technology, Jun. 2018. [Online]. Available: http://hdl.handle.net/11250/2564445

[10] F. Lang, "EXP.OPEN 2.0: A Flexible Tool Integrating Partial Order, Compositional, and On-the-fly Verification Methods," in *Proceedings of the 5th International Conference on Integrated Formal Methods (IFM'05), Eindhoven, The Netherlands*, ser. Lecture Notes in Computer Science, J. Romijn, G. Smith, and J. van de Pol, Eds., vol. 3771. Springer, Nov. 2005, pp. 70–88, full version available as INRIA Research Report RR-5673.

[11] P. Ledent, R. Mateescu, and W. Serwe, "Testing Resource Isolation for System-on-Chip Architectures," *Electronic Proceedings in Theoretical Computer Science*, vol. 399, pp. 129–168, Mar. 2024. [Online]. Available: http://dx.doi.org/10.4204/EPTCS.399.7

[12] L. Marsso, R. Mateescu, and W. Serwe, "TESTOR: A Modular Tool for On-the-Fly Conformance Test Case Generation," in *Proceedings of the 24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'18), Thessaloniki, Greece*, ser. Lecture Notes in Computer Science, D. Beyer and M. Huisman, Eds., vol. 10806. Springer, Apr. 2018, pp. 211–228.

[13] ——, "Automated Transition Coverage in Behavioural Conformance Testing," in *32nd IFIP Int. Conference on Testing Software and Systems (ICTSS'20), Naples, Italy*. Springer, 2020, pp. 219–235.

[14] R. Mateescu and D. Thivolle, "A Model Checking Language for Concurrent Value-Passing Systems," in *Proceedings of the 15th International Symposium on Formal Methods (FM'08), Turku, Finland*, ser. Lecture Notes in Computer Science, J. Cuellar, T. Maibaum, and K. Sere, Eds., vol. 5014. Springer, May 2008, pp. 148–164.

[15] J. Nagar, T. Dworzak, S. Simon, U. Heinkel, and D. Lettnin, "MetaPSS: An Automation Framework for Generation of Portable Stimulus Model," in *DVCon Europe 2023; Design and Verification Conference and Exhibition Europe*, 2023, pp. 79–85.

[16] Portable Stimulus Working Group, "Portable Test and Stimulus Standard 2.1," Accellera Systems Initiative, Elk Grove, CA, USA, Accellera standards, Oct. 2023. [Online]. Available: https://accellera.org/images/downloads/standards/pss/Portable_Test_Stimulus_Standard_v2.1.pdf

[17] H. Witharana, Y. Lyu, S. Charles, and P. Mishra, "A survey on assertion-based hardware verification," *ACM Comput. Surv.*, vol. 54, no. 11s, sep 2022.