Testing Resource Isolation for System-on-Chip Architectures

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

Ensuring resource isolation at the hardware level is a crucial step towards more security inside the Internet of Things. Even though there is still no generally accepted technique to generate appropriate tests, it became clear that tests should be generated at the system level. In this paper, we illustrate the modeling aspects in test generation for resource isolation, namely modeling the behavior and expressing the intended test scenario. We present both aspects using the industrial standard PSS and an academic approach based on conformance testing.

1 Introduction

SoC (System-on-Chip) architectures are being designed and deployed as microcontrollers of embedded systems. An SoC is usually highly configurable in order to perform several specific tasks in numerous devices, including smartphones or objects in the IoT (Internet of Things). SoC security is gaining importance as SoCs become ubiquitous, notably because they are being specially manufactured for heavy usage of machine learning and artificial intelligence in the IoT. Considering the distributed nature of the IoT, software solutions to security are insufficient, because an attacker can easily gain access to some hardware and tamper with it. Hardware attacks consist in forcing an SoC to perform operations in order to access functionalities or information that should normally not be available. A critical security requirement is *resource isolation*, which forbids applications (or programs) running on a same SoC to access data not intended for them.

Ensuring this requirement at hardware level is hence becoming mandatory to strengthen security, but is complex and still leaves two challenging problems. First, there is yet no commonly accepted solution: [16] claims to have found a side channel attack that might be applicable to any microcontroller and enable an attacker to access data from secure memory. Second, there is yet no commonly accepted approach for validating a proposal for a hardware resource isolation solution: most research focuses on attacking hardware implementations instead of formally validating proposed protocols.

When it comes to IoT devices, microcontroller manufacturers use the ARM Platform Security Architecture[1] which comes with a security specification and the possibility of certification by ARM (PSA-Certified[2]). The ARM Security Models [2] is the open-source ARM architecture for IoT with security concerns. Here, we focus on the resource isolation aspects of the ARM Security Models that are implemented with the notions of *security* (TrustZone [3]) and *privilege* (TrustZone alone not being enough [12]). ARM provides the possibility to carry security and privilege over the hardware through signals of its AMBA communication protocols [1] between a source and a target component. Filtering properly this information can then be left to the target or a dedicated component on the way in charge of monitoring the communication.

*Institute of Engineering Univ. Grenoble Alpes

[1] https://newsroom.arm.com/news/psa-next-steps-toward-a-common-industry-framework-for-secure-iot

[2] https://www.psacertified.org/

F. Lang, M. Volk (Eds):
Models for Formal Analysis of Real Systems (MARS 2024)
EPTCS 399, 2024, pp. 129–168, doi:10.4204/EPTCS.399.7

Before certifying an SoC by ARM, industrial manufacturers are concerned about representing and testing resource isolation for themselves (the case study [5] showed that ARM-Certified Level 2 may *leak* confidential information such as AES encryption keys). Resource isolation should ensure that data contained in an IP (Intellectual Property, as are components usually called in the hardware community) protected with given security and privilege levels can only be accessed by an IP with corresponding or higher levels. This kind of requirement can be checked using classical tools and techniques for industrial verification, such as hardware simulators using directed tests and/or execution-time assertions. Although properly written assertions are perfect to monitor exactly the behavior of a design under test during a simulation, it is still necessary to generate appropriate test scenarios to be executed: on its own, assertion-based verification cannot generate such scenarios.

The terrifying complexity of modern SoCs pushes to represent and reason about SoC behavior at higher abstraction levels, to ease the fast generation of many tests. For this purpose, PSS (the Portable-test Stimulus Standard) [15] was published by the Accellera Consortium[3] that comprises manufacturers such as AMD, ARM, Intel, Nvidia, NXP, and STMicroelectronics, but also major CAD tool vendors, such as Cadence, Siemens EDA, and Synopsis. PSS aims at providing an easy way to generate (many) tests, without the prior need to explicitly model too much of the SoC's behavior. PSS defines a (programming) language to abstract the behavior of an SoC as a set of "*actions*", which communicate and interact through "*flow objects*". PSS also defines a methodology to generate tests from a VI ("*Verification Intent*", a test scenario given as a partial ordering of the actions) by filling any gaps of the VI with appropriate actions, meeting the ordering constraints expressed for the SoC. Industrial manufacturers are inclined to use PSS, because it uses a familiar syntax (close to C++) and is well integrated in their current design flow and tools.

Although PSS has the appearance of a model-based testing approach, the emphasis is clearly more on the test generation, trying to minimize the time spent on the modeling. Furthermore, because there is no formal semantics of PSS, nor a complete definition of the underlying behavior corresponding to the set of constraints describing an SoC in PSS, the tasks of verification engineers remain difficult. The major challenge faced by these PSS users is getting a grasp on the behavior used as basis for test generation. Frequently, an erroneous constraint is only detected when an unexpected test is generated, limiting the confidence in the quality and coverage of the generated tests.

In this paper, we compare the modeling-related aspects of two approaches for test-case generation, namely the PSS approach with an approach based on conformance test generation with test purposes [10] as supported by the CADP toolbox [7] and its modeling language LNT [8]. Both approaches involve two separate modeling tasks: coming up with an abstract model of the SoC's behavior and expressing the structure of the desired test scenarios. However, the focus of both approaches is different: conformance testing starts with a model, whereas PSS favors modeling the test scenarios. This reflects the needs of verification engineers in the hardware design industry: at the end of the day, they have to produce tests for the SoC, and modeling is is acceptable only if it serves this purpose. We also study the impact of the difference in focus on the generated test suites.

We illustrate both approaches on the problem of generating tests for resource isolation, using a model of an SoC where the details of the various bus communication protocols are abstracted (each transaction is represented by a single rendezvous), because their differences and details are irrelevant to the test case generation. For both approaches, we separately discuss the modeling challenges concerning the behavior of the SoC and the structure of the test scenarios.

Formal verification is slowly being integrated in SoC design and verification workflows as shown in

[3] `https://accellera.org`

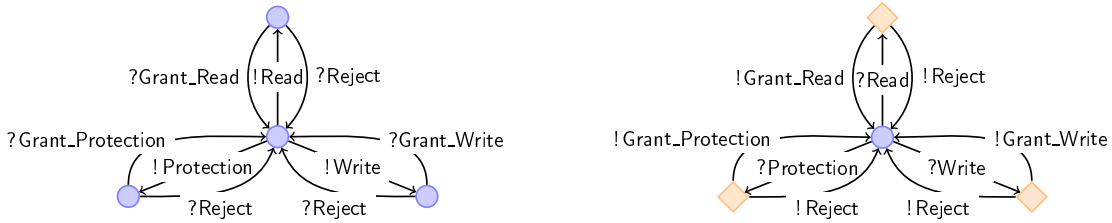

Figure 1: Symbolic automata representation of source (left) and target (right) behaviors

survey [9] but not for testing resource isolation. The closest work to our approach is [4] which proposes a high-level model of Intel 64 and ARMv8-A architectures to compare them but it neither formally specifies the behavior nor is the model used as basis for test generation.

The rest of this paper is organized as follows. Section 2 presents and compares several models of the resource-isolation related SoC behavior in LNT and PSS. Section 3 presents the modeling of test scenarios as test purposes in LNT and verification intents in PSS, together with the resulting test suites (sets of generated tests). Section 4 concludes. The complete LNT and PSS code is given in the appendices and provided in the MARS model repository.

2 Modeling the SoC Behavior for Resource Isolation

We illustrate resource isolation on an SoC with two kinds of IPs (components): sources (e.g., a CPU) and targets (e.g., a memory or dedicated hardware component storing sensible data). All IPs communicate through a bus-like shared interconnect, which can handle a single transaction at a time.[4] Both source and target have a *security level* (secure and non-secure) and a *privilege level* (privileged and non-privileged). Each target stores a data (data1 or data2). Each source can execute transactions to read or write the data of a target, or change the security and/or privilege levels of the data stored by the target. Each transaction consists of an access request emitted by the source, followed by a response (grant or reject) from the target. Each request by the source to the target includes the security and privilege levels of the source, as is the case for any AMBA [1] conform hardware protocol. The target should grant a read or write access if and only if both the security and privilege level of the source are at least those of the target. Concretely, a read or write request is rejected in the two (non-exclusive) situations where the target is secure (respectively, privileged) and the source is not. Changing the security and/or privilege level of the target is only granted to a secure and privileged source. We also allow the source to change its configuration (data to be written and security and privilege levels), thus including the case where a source (CPU) executes applications with different security and privilege levels.

Figure 1 shows the behaviors of a source (on the left) and a target (on the right) as two communicating symbolic automata, focusing on the executed actions and hiding all concrete data as well as security and privilege levels.[5] Both automata synchronize their transitions with identical labels; we omitted the transition corresponding to a change of the source configuration, because it is the only unsynchronized

[4]There are more complex communication protocols enabling a source to initiate further transactions with other targets, but this requires more than one interconnect.

[5]Taking them into account would yield unreadable figures: for instance, the source automaton would have as many different central (idle) states as there are different combinations of security and privilege levels (i.e., sixteen). See also the size of the LTS of the LNT model given in Sect. 2.1.

transition.

All requests are *initiated* by the source (marked with an exclamation-mark "!") and *received* by the target (marked with a question-mark "?"). The situation is the opposite for granting or rejecting a request (initiated by the target and received by the source). Initially, both automata are in their central state, the source is secure and privileged, and the target is non-secure and non-privileged.

The source can attempt a Read, a Write, or change the Protection level of the target; the target responds by Granting or Rejecting the request depending of whether or not it was legal. For each transaction, the source states its security and privilege levels and moves to the state corresponding to its request, where it awaits a response (grant or reject) from the target, before it can issue the next request.

The target starts by awaiting a transaction request from the source. After reception it enters a state represented by a lozenge symbol indicating that it must analyze the request to determine whether the request will be granted or rejected. This decision constitutes the resource isolation. A request is rejected for security reasons if and only if a non-secure source attempts to access a secure target. Likewise, a request is rejected for privilege reasons if and only if a non-privileged source attempts to access a privileged target. Altogether, a request is accepted if and only if it is not rejected for either reason. Only a secure and privileged request can change the security and privilege levels of the target. Note that the target accepts a new request only in its central state, thus the target accepts a new request only *after* having generated a grant or reject of the pending request (if any).

2.1 SoC Behavior Modeling in LNT

Such an SoC can be expressed easily using LNT [8], a modern language combining a sound foundation in concurrency theory with user-friendly syntax akin to mainstream programming languages. Two LNT processes define the behavior of a source or target, each encoding the corresponding symbolic automaton as an infinite loop, each iteration of which selects among the various possible actions. The overall model of the SoC is obtained as a parallel composition of an instance of as many sources and targets as there are in the SoC. Each communication on the interconnect is modeled as a multiway rendezvous between all instances (reflecting the fact that all IPs can observe everything exchanged on the interconnect). The complete LNT model (about 200 lines) is given in Appendix A.

The LNT model defines four enumerated data types: the security (security) and privilege (privilege) levels[6], the available data values (data), and the identities of the various IPs (ip). For the latter, the model also defines a function source(id) returning **true** if and only if id identifies a source IP. The function valid_access (s, t, p, q) returns **true** if and only if a source with security level t and privilege level q should be granted the request to read or write the data stored in a target with security level s and privilege level p.

Figure 2 shows the LNT process TARGET modeling a target. TARGET has a variable parameter id identifying the IP; the **require**-clause of line 4 enforces that this IP is indeed a target. Among the ten local variables, three record the currently stored data (d), security (s), and privilege (p). The other local variables serve to collect values exchanged during the rendezvous, so as to impose constraints (e.g., that the IP emitting a request is a source or whether the request should be granted or rejected based on the security and privilege level of source and target) or handle data (e.g., change the stored data on line 18 or the security and privilege levels on line 25). Each request is represented by a rendezvous on the corresponding gate (Read, Write, or Protection), during which the source transmits its current security and privilege levels, which the target stores in its local variables t and q (this is indicated by the question

[6]Without loss of generality, we restrict the model to two privilege levels (rather than the four considered by ARM).

```
process TARGET [Read, Grant_Read, Reject_Read, Write, Grant_Write,
                Reject_Write, Protection, Grant_Protection,
                Reject_Protection: Bus] (id: ip) is
   require not (source (id));
   var d,e: data, s,t,u: security, p,q,r: privilege, o, other: ip in
      d := data1; -- default value
      s := non_secure; p := non_privileged; -- lowest protection level
      loop
         select
            Read (?o, id, ?t, ?q) where source (o);
            if valid_access (s, t, p, q) then
               Grant_Read (o, id, d)
            else
               Reject_Read (o, id)
            end if
         [] Write (?o, id, ?t, ?q, ?e) where source (o);
            if valid_access (s, t, p, q) then
               d := e;
               Grant_Write (o, id)
            else
               Reject_Write (o, id)
            end if
         [] Protection (?o, id, ?t, ?q, ?u, ?r) where source (o);
            if (t == secure) and (q == privileged) then
               s := u; p := r;
               Grant_Protection (o, id, s, p)
            else
               Reject_Protection (o, id)
            end if
         -- communication between other IPs on the shared interconnect
         [] Read (?other, ?o, ?any security, ?any privilege)
               where (o != id) and source (other)
         [] Grant_Read (?other, ?o, ?any data)
               where (o != id) and source (other)
         [] Reject_Read (?other, ?o)
               where (o != id) and source (other)
         [] Write (?other, ?o, ?any security, ?any privilege, ?any data)
               where (o != id) and source (other)
         [] Grant_Write (?other, ?o)
               where (o != id) and source (other)
         [] Reject_Write (?other, ?o)
               where (o != id) and source (other)
         [] Protection (?other, ?o, ?any security, ?any privilege,
                        ?any security, ?any privilege)
               where (o != id) and source (other)
         [] Grant_Protection (?other, ?o, ?any security, ?any privilege)
               where (o != id) and source (other)
         [] Reject_Protection (?other, ?o)
               where (o != id) and source (other)
         end select
      end loop
   end var
end process
```

Figure 2: LNT process of a target

marks ? in lines 10, 16, 23, etc.). Depending on the validity of the request, the latter is either granted or rejected (by a rendezvous on the corresponding gate). For a Write and Protection, the grant is preceded by an update of the local variables of the target with the values received from the source during the request (see lines 18 and 25).

The LTS corresponding to a parallel composition of eight sources (which can only initiate the three transactions Read, Write and Protection) and a single target can be generated in less than a minute, and has 182 states, 558 transitions, and 99 labels (after minimization modulo strong bisimulation).

In a second version of the LNT model, a source not engaged in a transaction can also change its configuration (the data written by the source and the security and privilege level of the source). This corresponds to considering sources as multitasking-enabled CPUs capable of executing several applications with different configurations, and to take care of the configuration changes induced by switching between applications. The LTS corresponding to this extended model is too large to be generated—the number of states is expected to be 8^8 times the size of the previous model. However, when removing the identification of the source IP from all transition labels and hiding all transitions corresponding to a configuration change, both LTSs are equivalent for branching bisimulation (the LTS minimized modulo branching bisimulation has 52 states, 268 transitions, and 39 labels).

The identity of the source IP seems thus not important. Indeed, when removing the identification of the source IP from all transition labels and hiding all transitions corresponding to a configuration change, a model with a single multitasking-enabled source also is equivalent for branching bisimulation to the model with eight sources that do not have multitasking enabled. Hence, with the possibility to change the source configuration, it is sufficient to model a single source.

The situation is more intricate concerning the number of targets. Actually, two targets are independent and thus equivalent to a single target with two memory cells with separate security and privilege levels. However, resource isolation is concerned with the access to a single target, so that it is not necessary to study SoCs with more than one target.

It is worth mentioning that the LTS can be analyzed with a full range of verification tools, e.g., those provided by the CADP toolbox. Besides the equivalence checking tools already used to compare the SoCs with different numbers of sources, it is possible to explore the LTS step by step and to verify temporal logic properties. This is helpful to gain confidence in the correctness of the modeled behavior.

2.2 SoC Behavior Modeling in PSS

A major modeling difference between LNT and PSS is that LNT is targeted at modeling the SoC, whereas PSS avoids modeling the overall behavior of the SoC, focusing on simply expressing constraints between the actions of the SoC. However, the latter is less convenient when it comes to precisely understand the modeled behavior, because it requires to assemble all these constraints together.

The understanding of the behavioral model induced by the constraints can be improved by adopting a modeling discipline, such as encoding the two symbolic automata of Fig. 1 (as seen in the previous section, it is sufficient to consider an SoC with a single source and a single target). For each automaton, each transition can be encoded as a PSS action, which inputs from and outputs to a (same) state flow object storing the data values of the automaton, using constraints to enable actions only for particular states of the automaton and controlling the state resulting from the execution of an action. Synchronization between the automata is then expressed using stream flow objects, mimicking the multiway rendezvous on the gates in the LNT model.

This intuitive approach yields the PSS model presented in Appendix B, featuring two state flow objects, nine stream flow objects, and a total of 21 actions (ten actions for the transitions of the source, nine

actions for the transitions of the target, plus two actions to control the initial state of the two state flow objects—this is required by the PSS semantics). This significant increase in complexity is accompanied by the need to specify for all actions not only the fields of the state flow object that are modified, but also those that remain unchanged. All in all, the corresponding PSS model ends up with more than 500 lines.

It is possible to translate this PSS model to LNT (using a translator currently under development), leading to almost two thousand lines of LNT. This generic translation encodes each action and flow object as a separate LNT process, leading to a total of 32 processes. The LTS corresponding to each of these processes can be generated and minimized modulo divergence-preserving branching bisimulation, before composing all 32 LTSs into the overall LTS of the PSS model.[7] This generation of the corresponding LTS took about a day (on the yeti cluster in the Grenoble site of the Grid'5000 platform), exploiting the 64 cores using a distributed state space generation tool. However, the corresponding state space (before hiding all transitions related to the interactions between actions and flow objects) is prohibitively large: 1,700,860,640 states, 13,934,786,272 transitions, and 6,706 labels, stored in a file with a size of 88 GB. Note that more refined compositional generation strategies (e.g., smart generation [6]) did not succeed, as some intermediate state spaces for a subset of the processes are larger than the overall state space.

Taking into account that a rendezvous between several actions yields a unique visible transition, we investigated a simpler modeling approach encoding a monolithic automaton, incorporating the constraints of both source and target. This approach requires only ten actions (three requests, three grants, three rejects, and the configuration change), all inputting from and outputting to a single state flow object. The corresponding PSS code is given in Appendix C. The drawback of this approach is the increase in constraints for each action, because it is necessary to specify all fields of the state flow that remain unchanged by the action (each field related to the target is not affected by an action related to the source and vice-versa). Another inconvenient of this approach is that it would be very impractical to extend this model to an SoC with more IPs, due the complexity of getting a complete and correct set of constraints.

This monolithic PSS model can also be translated into (almost one thousand lines of) LNT, from which the corresponding LTS (2736 states, 4591 transitions, and 4592 labels) can be directly[8] generated in less than a minute. After hiding all transitions related to interactions with the state flow object, changing all transition labels to use the same gates and sets of offers as the LNT models of the previous section, and determinization (reduction for weak trace equivalence), the LTS is branching equivalent to those of the LNT models presented in the previous section.

Figure 3 gives the description of action `target_grant_read`. It inputs from and outputs to a state flow object, which keeps track of the configuration of the SoC. Execution of the action is subject to the **constraint**s specified in its body. The first constraint (line 5) enforces that the action can be executed only if another action has already output to the state flow object (each PSS state flow object has an implicit field `initial`, which is initialized to **true**, changed to **false** upon the first output to the flow object, and never changed again). The next two constraints express that the source automaton moves from `read` (line 7 constraining the value of field `sstate` of the input flow object `in_state`) back to `idle` (line 8 constraining field `sstate` of the output flow object `out_state`). The next two constraints (lines 10–13) express the validity of the transaction (inspecting only fields of the input flow object). The remaining eight constraints express that all other fields of the output flow object should keep the values of the fields of the input flow object.

This 24-line PSS description of the action (with its constraints) is more verbose than the correspond-

[7]The translation of stream flow objects makes use of the n-among-m synchronization currently only supported by the EXP.OPEN [11] tool.

[8]Due to the absence of stream flow objects, the generated LNT model does not require a n-among-m synchronization and can thus be handled directly by the LNT compiler.

```
1   action t_grant_read {
2     input  system_state in_state;
3     output system_state out_state;
4
5     constraint in_state.initial == false;
6     // Move from Read to Idle
7     constraint in_state.sstate  == read;
8     constraint out_state.sstate == idle;
9     // Check protection
10    constraint (in_state.source_sec == secure) ||
11               (in_state.target_sec == non_secure);
12    constraint (in_state.source_priv == privileged) ||
13               (in_state.target_priv == non_privileged);
14    // Maintain source fields
15    constraint out_state.source_sec  == in_state.source_sec;
16    constraint out_state.source_priv == in_state.source_priv;
17    constraint out_state.source_data == in_state.source_data;
18    // Maintain target fields
19    constraint out_state.target_sec  == in_state.target_sec;
20    constraint out_state.target_priv == in_state.target_priv;
21    constraint out_state.target_data == in_state.target_data;
22    constraint out_state.new_sec     == in_state.new_sec;
23    constraint out_state.new_priv    == in_state.new_priv;
24  }
```

Figure 3: Action for granting a read request in the monolithic PSS model

ing three lines of LNT (lines 13–15 in Fig. 2). This has several reasons. First, in PSS the states of the target have to be listed explicitly, whereas they are deduced from the control flow in LNT. Second, LNT has no implicit field `initial`. Last, but not least, in LNT it is not necessary to specify the variables that maintain their value.

3 Test Generation from Test Scenarios

The principal objective of the models of the SoC behavior presented in Section 2 is to enable the generation of tests to validate the SoC. Characterizing a set of desired tests is a modeling task of its own, based on the idea of expressing a partial ordering of some actions that have to appear in the generated tests, and of relying on tools exploiting the behavioral model to fill in any further actions necessary to obtain a complete test case. This approach emphasizes the expression of a *test scenario* defining the high-level structure of the tests, leaving the details to automatic tools. The notion of test scenario is called TP (*test purpose*) in conformance testing theory [10] and VI (*verification intent*) in PSS.

There are different techniques to construct tests from a test scenario. The TESTOR tool [13] proceeds by a *forward exploration* of a (particular) synchronous product between the TP and the behavioral model, extracting on-the-fly a test or a subgraph called CTG (complete test graph) containing all possible tests for the TP. The PSS methodology [15, Appendix F] uses a *backward traversal* of the VI, determining for each action its immediately necessary previous actions, based on the constraints in the verification intent and the behavioral model. In the following, we compare the effect of these different approaches on four

```
1  process PURPOSE_1 [
2              Reject_Read,
3              Reject_Write,
4              Reject_Protection,
5              TESTOR_ACCEPT: none] is
6     select
7        Reject_Read
8     [] Reject_Write
9     [] Reject_Protection
10    end select;
11    loop TESTOR_ACCEPT end loop
12 end process
```

```
action intent_1 {
  t_reject_read       Reject_Read;
  t_reject_write      Reject_Write;
  t_reject_protection Reject_Protection;
  activity {
    select{
        Reject_Read;
        Reject_Write;
        Reject_Protection;
    }
  }
}
```

Figure 4: Test scenario 1 (“reject for any reason”) as TP in LNT (left) and VI in PSS (right)

test scenarios for resource isolation.

3.1 Test Scenario 1: Reject for any Reason

A natural first test scenario for resource isolation is to search for tests featuring the detection of an illegal transaction, i.e., containing any of the three actions Reject_Read, Reject_Write, and Reject_Protection. Figure 4 shows how to express this scenario as a TP in LNT and a VI in PSS.

In LNT the TP is encapsulated in a process PURPOSE_1, the gate parameters (lines 2–5) of which are the three actions expected in the scenario plus the special gate TESTOR_ACCEPT indicating the goal of the TP. The behavior of this TP is the sequential composition of a non-deterministic choice (**select** instruction in lines 6–10, choices being separated by “**[]**”) among the three actions, followed by a loop indicating the end of the TP.

In PSS the VI is a compound action, referencing the three actions via action handles (lines 2–4). The ordering of actions is specified by the **activity** block (lines 5–11), containing a non-deterministic **select**ion among the three actions (lines 6–10, choices being separated by “**;**”).

For this TP, TESTOR generates a CTG (183 states, 567 transitions, and 101 labels) that contains all paths to reach any of the three actions, including paths with granted requests before the rejected one. A CTG can be considered a description of a tester, interacting with the SoC to drive it towards the goal of the TP, by selecting appropriate control actions (or inputs) depending on the outputs observed so far. In general, a CTG contains states, where the tester has to choose among different control actions to be executed. The CTG generated for this TP contains 384 choices, all of which can be covered by a suite of 357 test cases that can be generated automatically using the approach proposed in [14].

For this VI, the PSS backward traversal starts by (non-deterministically) choosing one of the three reject actions, and then determines which other actions must immediately precede, by checking which action could have written values to the state flow object so as to satisfy the input constraints of the selected action. The constraints on the sstate field imply the preceding action must be a request. For Reject_Read and Reject_Write, the constraints on the security and privilege levels imply that in the request, one of these values must be strictly lower than the one of the target. For the Reject_Protection, the constraints imply that the preceding Request_Protection stems from a source that is not both secure and privileged. For the monolithic behavioral model, this backward traversal continues until the action init_system_state

```
process PURPOSE_2 [
            Reject_Read,
            Reject_Write,
            Reject_Protection,
            Grant_Read,
            Grant_Write,
            Grant_Protection,
            TESTOR_ACCEPT: none] is
   par
      Grant_Read
   || Grant_Write
   || Grant_Protection
   || Reject_Read
   || Reject_Write
   || Reject_Protection
   end par;
   loop TESTOR_ACCEPT end loop
end process
```

```
action intent_2 {
  t_grant_read          Grant_Read;
  t_grant_write         Grant_Write;
  t_grant_protection    Grant_Protection;
  t_reject_read         Reject_Read;
  t_reject_write        Reject_Write;
  t_reject_protection   Reject_Protection;
  activity {
    schedule{
        Grant_Read;
        Grant_Write;
        Grant_Protection;
        Reject_Read;
        Reject_Write;
        Reject_Protection;
    }
  }
}
```

Figure 5: Test scenario 2 ("all possible responses" interleaved) as TP in LNT (left) and VI in PSS (right)

is found.[9] In practice, the PSS methodology aims at generating a single test at each invocation. When implemented using a breadth-first backward traversal (as is the case for some industrial PSS tools), this systematically yields any of the shortest possible tests.

3.2 Test Scenario 2: Test all Possible Responses (Interleaving Semantics)

This test scenario aims at observing all responses to the three transactions, in any order using the *interleaving* of the responses as shown in Figure 5. In LNT, the parallel composition operator **par** expresses the interleaving of the different branches separated by "||". In PSS, the **schedule** operator expresses the interleaving of the branches separated by ";".[10]

For this TP, TESTOR computes a CTG with 2649 states and 12,057 transitions; its 8832 choices can be covered with 8328 tests. The size of the CTG is due to the fact that once one of the responses has been observed, it is still possible to observe it before all responses have been observed. Hence, the CTG corresponds to an "unfolding" of the model six times, repeating the complete behavior of the SoC until all responses have been observed.

Searching for short(est) tests, the PSS methodology reduces the number of changes in the security and privilege levels of the source and the target. Therefore, in most tests the security and privilege levels for Grant_Read and Grant_Write (respectively Reject_Read and Reject_Write) are the same, and Grant_Protection and Reject_Protection are inserted where suitable. Notice that the syntactic order of the responses in the VI (and TP) actually corresponds to the shortest sequence. Indeed, because the model starts with a secure and privileged source and a non-secure and non-privileged target, all grants are possible. Increasing the security and/or privilege of the target and appropriately lowering the security and privilege of the source are then sufficient to observe the three rejections.

[9] For the generic PSS behavioral model, both init_source_state and init_target_state have to be found.

[10] The PSS operator **parallel** expresses a parallel execution of different behaviors using several threads.

```
1  process PURPOSE_3 [
2              Reject_Read,
3              Reject_Write,
4              Reject_Protection,
5              Grant_Read,
6              Grant_Write,
7              Grant_Protection,
8              TESTOR_ACCEPT: none] is
9     Grant_Read;
10    Grant_Write;
11    Grant_Protection;
12    Reject_Read;
13    Reject_Write;
14    Reject_Protection;
15    loop TESTOR_ACCEPT end loop
16 end process
```

```
action intent_3 {
  t_grant_read          Grant_Read;
  t_grant_write         Grant_Write;
  t_grant_protection    Grant_Protection;
  t_reject_read         Reject_Read;
  t_reject_write        Reject_Write;
  t_reject_protection   Reject_Protection;
  activity {
    Grant_Read;
    Grant_Write;
    Grant_Protection;
    Reject_Read;
    Reject_Write;
    Reject_Protection;
  }
}
```

Figure 6: Test scenario 3 (“all possible responses” in sequence) as TP in LNT (left) and VI in PSS (right)

3.3 Test Scenario 3: Test all Possible Responses (Sequential Semantics)

Most test generation strategies do not support the interleaving of actions, but require more *directed* specifications enforcing a particular sequence of actions. Test scenario 3 requests once again all possible responses but in a particular order, expressed in LNT and PSS using “;”, as illustrated on Figure 6.

Requesting such a *directed* scenario has consequences on the generated test suite for both LNT and PSS. The CTG generated by TESTOR will contain for two sequential actions of the TP every possible path of the model allowed in between. The CTG has 967 states and 3271 transitions; its 2208 choices can be covered with 2072 tests. This CTG is smaller than the one for test scenario 2, because only a single ordering of responses is requested.

The tests generated by PSS are once again the shortest ones and included in those generated for test scenario 2. This shows that more directed test scenarios limit the set of generated tests.

3.4 Test Scenario 4: Access Data with Different Protection

Using the notions of security and privilege, ARM-PSA diversifies the different levels of protection possible for an IP in an SoC. However, there is the strong assumption of a *trusted administrator* as all requests of a secure and privileged source are necessarily granted. Test scenario 4 expresses that whatever the security and privilege of the target, a source with the same security and privilege can write to the target, and any source with higher security and/or privilege (e.g., the administrator) will be able to read the written data. This scenario requires to express that there should be no change in the security or privilege between the write and read requests.

Test Scenario 4 focuses on how to express the refusal of some behavior. This is illustrated in Figure 7 by describing a corresponding TP in LNT, using the special gate TESTOR_REFUSE (line 8) to indicate that the preceding rendezvous on gate Grant_Protection should be excluded from the generated CTG. The null branch (line 10) of the **select** construct (lines 6–11) allows any other action. The where clause on line 12 guarantees (in combination with the condition on line 11 of Figure 2) that the final read is requested with higher security and/or privilege than the write on line 5.

```
process PURPOSE_4 [Read, Grant_Read, Write, Grant_Protection: Bus,
                   TESTOR_ACCEPT, TESTOR_REFUSE: none] is
   var s,t: security, p,q: privilege, d: data in
      Grant_Protection (?any ip, ip0, ?s, ?p)
      Write (?any ip, ip0, s, p, ?d); -- same s and p as in the previous line
      select
         -- refuse any further rendezvous on gate Grant_Protection
         Grant_Protection (?any ip, ip0, ?s, ?p); loop TESTOR_REFUSE end loop
      [] -- accept all other rendezvous
         null
      end select;
      Read (?any ip, ip0, ?t, ?q) where (s != t) or (p != q);
      Grant_Read (?any ip, ip0, d); -- access data with different security and privilege levels
      loop TESTOR_ACCEPT end loop
   end var
end process
```

Figure 7: Test scenario 4 (“access data with different security/privilege”) as TP in LNT

To the best of our knowledge, PSS has no such means to explicitly request absence of actions from the generated tests. Instead, the scenario has to be made more directed by explicitly including more actions in the VI so as to add constraints on these actions. In particular, the VI allows to **bind** an input flow object of an action a_2 to the output flow object of another action a_1, constraining action-inference and forcing a_1 to immediately precede a_2. The resulting, lengthy VI is given in Appendix C.

4 Conclusion

In this paper, we illustrated the modeling tasks for testing hardware resource isolation using both the approach promoted by the industrial standard PSS and an academic approach based on LNT and conformance testing. Both approaches require a model of behavior and an abstract test scenario, which is refined into concrete tests based on the behavioral model.

Despite these similarities, both approaches differ in the way of generating tests, using a forward (LNT) or backward (PSS) search. This difference not only yields different tests, but also impacts the modeling, due to the trade-off between putting constraints in the behavior model or the test scenario. On the one hand, LNT facilitates a complete, verifiable model of the behavior, from which extensive test suites can be generated with few, short test scenarios. On the other hand, PSS favors focusing on the test scenario (or verification intent), and requires longer test scenarios to obtain longer tests. While this avoids the risk of state space explosion, it comes at the price of losing the coverage guarantees available for conformance testing, in particular in the presence of cyclic behavior. Furthermore, the behavior is often under-constrained in PSS, especially when adding a new action to the behavior.

The models presented in this paper were used in an industrial context. An extended version of test scenario 3 requested in LNT a specific order of attempting each transaction for all combinations of source and target security and privilege levels. Concretely, for each attempted transaction, the source requests to write, to read, and then to change the target’s security and privilege. From the generated CTG, we derived a single long test (including all transaction attempts). This test was included in the

nightly non-regression tests for a (confidential) SoC under development, sequentially executing the test for each of the over hundred target IPs of the SoC. This revealed a few cases of bad wiring, unaligned documentation, and misunderstandings between architect, design, and verification engineers.

Because the behavioral model of PSS is hard to grasp, modeling errors are frequently detected only by the generation of unexpected tests. We are currently working on the automated translation of PSS constructs into LNT to support the early analysis of the behavioral model, e.g., by model checking. This also includes guidelines for devising PSS models with an efficient translation to LNT.

References

[1] ARM: *AMBA Specification (Rev 2.0)*. Available at `https://developer.arm.com/documentation/ihi0011/a`.

[2] ARM: *Platform Security Model 1.1*. Available at `https://developer.arm.com/documentation/den0128/latest`.

[3] ARM: *Security in an ARMv8 System*. Available at `https://developer.arm.com/documentation/100935/0100/Security-in-ARMv8-A-systems-`.

[4] Guillaume Averlant, Benoît Morgan, Éric Alata, Vincent Nicomette & Mohamed Kaâniche (2017): *An Abstraction Model and a Comparative Analysis of Intel and ARM Hardware Isolation Mechanisms*. In: *2017 IEEE 22nd Pacific Rim International Symposium on Dependable Computing (PRDC)*, pp. 245–254, doi:10.1109/PRDC.2017.48.

[5] Fei Chen, Duming Luo, Jianqiang Li, Victor C. M. Leung, Shiqi Li & Junfeng Fan (2023): *Arm PSA-Certified IoT Chip Security: A Case Study*. *Tsinghua Science and Technology* 28(2), pp. 244–257, doi:10.26599/TST.2021.9010094.

[6] Pepijn Crouzen & Frédéric Lang (2011): *Smart Reduction*. In Dimitra Giannakopoulou & Fernando Orejas, editors: *Proceedings of Fundamental Approaches to Software Engineering (FASE'11), Saarbrücken, Germany*, *Lecture Notes in Computer Science* 6603, Springer, pp. 111–126, doi:10.1007/978-3-642-19811-3_9.

[7] Hubert Garavel, Frédéric Lang, Radu Mateescu & Wendelin Serwe (2013): *CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes*. *Springer International Journal on Software Tools for Technology Transfer (STTT)* 15(2), pp. 89–107, doi:10.1007/s10009-012-0244-z.

[8] Hubert Garavel, Frédéric Lang & Wendelin Serwe (2017): *From LOTOS to LNT*. In Joost-Pieter Katoen, Rom Langerak & Arend Rensink, editors: *ModelEd, TestEd, TrustEd – Essays Dedicated to Ed Brinksma on the Occasion of His 60th Birthday*, *Lecture Notes in Computer Science* 10500, Springer, pp. 3–26, doi:10.1007/978-3-319-68270-9_1.

[9] Tomás Grimm, Djones Lettnin & Michael Hübner (2018): *A Survey on Formal Verification Techniques for Safety-Critical Systems-on-Chip*. *Electronics* 7(6), doi:10.3390/electronics7060081.

[10] Claude Jard & Thierry Jéron (2005): *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)* 7(4), pp. 297–315, doi:10.1007/s10009-004-0153-x.

[11] Frédéric Lang (2005): *EXP.OPEN 2.0: A Flexible Tool Integrating Partial Order, Compositional, and On-the-fly Verification Methods*. In Judi Romijn, Graeme Smith & Jaco van de Pol, editors: *Proceedings of the 5th International Conference on Integrated Formal Methods (IFM'05), Eindhoven, The Netherlands*, *Lecture Notes in Computer Science* 3771, Springer, pp. 70–88, doi:10.1007/11589976_6. Full version available as INRIA Research Report RR-5673.

[12] Wenhao Li, Yubin Xia & Haibo Chen (2019): *Research on ARM TrustZone*. *GetMobile: Mobile Comp. and Comm.* 22(3), pp. 17–22, doi:10.1145/3308755.3308761.

[13] Lina Marsso, Radu Mateescu & Wendelin Serwe (2018): *TESTOR: A Modular Tool for On-the-Fly Conformance Test Case Generation*. In Dirk Beyer & Marieke Huisman, editors: *Proceedings of the 24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'18), Thessaloniki, Greece*, *Lecture Notes in Computer Science* 10806, Springer, pp. 211–228, doi:10.1007/978-3-319-89963-3_13.

[14] Lina Marsso, Radu Mateescu & Wendelin Serwe (2020): *Automated Transition Coverage in Behavioural Conformance Testing*. In: *32nd IFIP Int. Conference on Testing Software and Systems (ICTSS'20), Naples, Italy*, Springer, pp. 219–235, doi:10.1007/978-3-030-64881-7_14.

[15] Portable Stimulus Working Group (2001): *Portable Test and Stimulus Standard 2.0*. Accellera standards, Accellera Systems Initiative, Elk Grove, CA, USA. Available at https://accellera.org/images/downloads/standards/Portable_Test_Stimulus_Standard_v20.pdf.

[16] C. Rodrigues, D. Oliveira & S. Pinto (2024): *BUSted!!! Microarchitectural Side-Channel Attacks on the MCU Bus Interconnect*. In: *2024 IEEE Symposium on Security and Privacy (SP)*, IEEE Computer Society, Los Alamitos, CA, USA, doi:10.1109/SP54263.2024.00062.

A LNT Model

```
module model_8_1 with ==, != is

--------------------------------------------------

type security is secure, non_secure end type
type privilege is privileged, non_privileged end type

type data is data1, data2 end type

type ip is ip0, ip1, ip2, ip3, ip4, ip5, ip6, ip7, ip8 end type

--------------------------------------------------

channel Bus is
   (source, target: ip),
   (source, target: ip, d: data),
   (source, target: ip, s: security, p: privilege),
   (source, target: ip, s: security, p: privilege, d: data),
   (source, target: ip, s: security, p: privilege, t: security, q: privilege)
end channel

--------------------------------------------------

function valid_access (s,t: security, p,q: privilege): Bool is
   -- returns true iff a source with protection level (t,q) is
   -- allowed to access a target with protection level (s,p)
   return not (((s == secure)     and (t == non_secure))      or
               ((p == privileged) and (q == non_privileged)))
end function

--------------------------------------------------

function source (id: ip) : bool is
   -- returns true iff id is a source IP
   case id in
      ip0 -> return false
   |  any -> return true
   end case
end function

--------------------------------------------------

process SOURCE [Read, Grant_Read, Reject_Read, Write, Grant_Write,
                Reject_Write, Protection, Grant_Protection,
                Reject_Protection: Bus, Change_Source_Config: any]
               (id: ip, in var s: security, in var p: privilege,
                in var d: data, multitasking: Bool) is
   require source (id);
   var o, other: ip in
```

```
      loop
         select
            Read (id, ?o, s, p) where not (source (o));
            select
               Grant_Read (id, o, ?any data)
            [] Reject_Read (id, o)
            end select
         [] Write (id, ?o, s, p, d) where not (source (o));
            select
               Grant_Write (id, o)
            [] Reject_Write (id, o)
            end select
         [] Protection (id, ?o, s, p, ?any security, ?any privilege)
               where not (source (o));
            select
               Grant_Protection (id, o, ?any security, ?any privilege)
            [] Reject_Protection (id, o)
            end select
         [] only if multitasking then
               Change_Source_Config (id, id, ?s, ?p, ?d)
            end if
         -- communication between other IPs on the shared interconnectkbu
         [] Read (?o, ?other, ?any security, ?any privilege)
               where (o != id) and not (source (other))
         [] Grant_Read (?o, ?other, ?any data)
               where (o != id) and not (source (other))
         [] Reject_Read (?o, ?other)
               where (o != id) and not (source (other))
         [] Write (?o, ?other, ?any security, ?any privilege, ?any data)
               where (o != id) and not (source (other))
         [] Grant_Write (?o, ?other)
               where (o != id) and not (source (other))
         [] Reject_Write (?o, ?other)
               where (o != id) and not (source (other))
         [] Protection (?o, ?other, ?any security, ?any privilege,
               ?any security, ?any privilege)
               where (o != id) and not (source (other))
         [] Grant_Protection (?o, ?other, ?any security, ?any privilege)
               where (o != id) and not (source (other))
         [] Reject_Protection (?o, ?other)
               where (o != id) and not (source (other))
         end select
      end loop
   end var
end process

--------------------------------------

process TARGET [Read, Grant_Read, Reject_Read, Write, Grant_Write,
                Reject_Write, Protection, Grant_Protection,
                Reject_Protection: Bus] (id: ip) is
   require not (source (id));
```

```
   var d,e: data, s,t,u: security, p,q,r: privilege, o, other: ip in
      d := data1; -- default value
      s := non_secure; p := non_privileged; -- lowest protection level
      loop
         select
            Read (?o, id, ?t, ?q) where source (o);
            if valid_access (s, t, p, q) then
               Grant_Read (o, id, d)
            else
               Reject_Read (o, id)
            end if
         [] Write (?o, id, ?t, ?q, ?e) where source (o);
            if valid_access (s, t, p, q) then
               d := e;
               Grant_Write (o, id)
            else
               Reject_Write (o, id)
            end if
         [] Protection (?o, id, ?t, ?q, ?u, ?r) where source (o);
            if (t == secure) and (q == privileged) then
               s := u; p := r;
               Grant_Protection (o, id, s, p)
            else
               Reject_Protection (o, id)
            end if
         -- communication between other IPs on the shared interconnect
         [] Read (?other, ?o, ?any security, ?any privilege)
               where (o != id) and source (other)
         [] Grant_Read (?other, ?o, ?any data)
               where (o != id) and source (other)
         [] Reject_Read (?other, ?o)
               where (o != id) and source (other)
         [] Write (?other, ?o, ?any security, ?any privilege, ?any data)
               where (o != id) and source (other)
         [] Grant_Write (?other, ?o)
               where (o != id) and source (other)
         [] Reject_Write (?other, ?o)
               where (o != id) and source (other)
         [] Protection (?other, ?o, ?any security, ?any privilege,
                        ?any security, ?any privilege)
               where (o != id) and source (other)
         [] Grant_Protection (?other, ?o, ?any security, ?any privilege)
               where (o != id) and source (other)
         [] Reject_Protection (?other, ?o)
               where (o != id) and source (other)
         end select
      end loop
   end var
end process

--------------------------------------
```

```
process SOC [Read, Grant_Read, Reject_Read, Write, Grant_Write, Reject_Write,
             Protection, Grant_Protection, Reject_Protection: Bus,
             Change_Source_Config: any] (multitasking: Bool) is
   par Read, Grant_Read, Reject_Read, Write, Grant_Write, Reject_Write,
       Protection, Grant_Protection, Reject_Protection
   in
      SOURCE [...] (ip1, secure, privileged, data1, multitasking)
   || SOURCE [...] (ip2, secure, privileged, data2, multitasking)
   || SOURCE [...] (ip3, secure, non_privileged, data1, multitasking)
   || SOURCE [...] (ip4, secure, non_privileged, data2, multitasking)
   || SOURCE [...] (ip5, non_secure, privileged, data1, multitasking)
   || SOURCE [...] (ip6, non_secure, privileged, data2, multitasking)
   || SOURCE [...] (ip7, non_secure, non_privileged, data1, multitasking)
   || SOURCE [...] (ip8, non_secure, non_privileged, data2, multitasking)
   || TARGET [...] (ip0)
   end par
end process

--------------------------------------------

process SOC_2 [Read, Grant_Read, Reject_Read, Write, Grant_Write, Reject_Write,
               Protection, Grant_Protection, Reject_Protection: Bus,
               Change_Source_Config: any] is
   par Read, Grant_Read, Reject_Read, Write, Grant_Write, Reject_Write,
       Protection, Grant_Protection, Reject_Protection
   in
      SOURCE [...] (ip1, secure, privileged, data1, true)
   || TARGET [...] (ip0)
   end par
end process

--------------------------------------------

process MAIN [Read, Grant_Read, Reject_Read, Write, Grant_Write, Reject_Write,
              Protection, Grant_Protection, Reject_Protection: Bus,
              Change_Source_Config: any] is
   SOC [...] (false)
end process

--------------------------------------------

process PURPOSE_1 [Reject_Read, Reject_Write, Reject_Protection,
                   TESTOR_ACCEPT: none] is
   -- any reject
   select
      Reject_Read
   [] Reject_Write
   [] Reject_Protection
   end select;
   loop TESTOR_ACCEPT end loop
end process
```

```
--------------------------------------------------

process PURPOSE_2 [Grant_Read, Grant_Write, Grant_Protection, Reject_Read,
                   Reject_Write, Reject_Protection, TESTOR_ACCEPT: none] is
   -- any transaction (all possible outcomes) in any order
   par
      Grant_Read
   || Grant_Write
   || Grant_Protection
   || Reject_Read
   || Reject_Write
   || Reject_Protection
   end par;
   loop TESTOR_ACCEPT end loop
end process

--------------------------------------------------

process PURPOSE_3 [Grant_Read, Grant_Write, Grant_Protection, Reject_Read,
                   Reject_Write, Reject_Protection, TESTOR_ACCEPT: none] is
   -- any transaction (all possible outcomes) in a seqential order
   Grant_Read;
   Grant_Write;
   Grant_Protection;
   Reject_Read;
   Reject_Write;
   Reject_Protection;
   loop TESTOR_ACCEPT end loop
end process

--------------------------------------------------

process PURPOSE_4 [Read, Grant_Read, Write, Grant_Protection: Bus,
                   TESTOR_ACCEPT, TESTOR_REFUSE: none] is
   -- granted read with different security/privilege than the preceding write
   var s,t: security, p,q: privilege, d: data in
      Grant_Protection (?any ip, ip0, ?s, ?p);
      Write (?any ip, ip0, s, p, ?d);
      -- forbid any change of the security/privilege of the target
      select
         null
      [] Grant_Protection (?any ip, ip0, ?s, ?p);
         loop TESTOR_REFUSE end loop
      end select;
      Read (?any ip, ip0, ?t, ?q) where (s != t) or (p != q);
      Grant_Read (?any ip, ip0, d);
      loop TESTOR_ACCEPT end loop
   end var
end process

--------------------------------------------------
```

```
end module
```

B PSS Model

```
component pss_top {

  // ----------------------------------------
  // Types
  // ----------------------------------------

  enum data_e {
    data1, data2
  }

  enum security_e {
    secure, non_secure
  }

  enum privilege_e {
    privileged, non_privileged
  }

  // ----------------------------------------
  // Stream Flow Objects for Communication
  // (three streams per operation, for request, grant, and reject)
  // ----------------------------------------

  // streams for read

  stream request_read_stream {
    rand security_e  sec;  // security of the source requesting to read
    rand privilege_e priv; // privilege of the source requesting to read
  }
  pool request_read_stream request_read_stream_pool;
  bind request_read_stream_pool *;

  stream grant_read_stream {
    rand data_e data; // read data
  }
  pool grant_read_stream grant_read_stream_pool;
  bind grant_read_stream_pool *;

  stream reject_read_stream {}
  pool reject_read_stream reject_read_stream_pool;
  bind reject_read_stream_pool *;

  // streams for write

  stream request_write_stream {
    rand security_e  sec;  // security of the source requesting to write
    rand privilege_e priv; // privilege of the source requesting to write
```

```
  rand data_e      data; // data to be written
}
pool request_write_stream request_write_stream_pool;
bind request_write_stream_pool *;

stream grant_write_stream {}
pool grant_write_stream grant_write_stream_pool;
bind grant_write_stream_pool *;

stream reject_write_stream {}
pool reject_write_stream reject_write_stream_pool;
bind reject_write_stream_pool *;

// streams for setting the protection

stream request_protection_stream {
  rand security_e  sec;       // security of the requesting source
  rand privilege_e priv;      // privilege of the requesting source
  rand security_e  next_sec;  // new security
  rand privilege_e next_priv; // new privilege
}
pool request_protection_stream request_protection_stream_pool;
bind request_protection_stream_pool *;

stream grant_protection_stream {}
pool grant_protection_stream grant_protection_stream_pool;
bind grant_protection_stream_pool *;

stream reject_protection_stream {}
pool reject_protection_stream reject_protection_stream_pool;
bind reject_protection_stream_pool *;

// ------------------------------------
// Finite State Machine for the Source
// ------------------------------------

enum source_state_e {
  idle, read, write, change
}

state source_state {
  rand source_state_e sstate;
  rand data_e      data;
  rand security_e  sec;
  rand privilege_e priv;
}
pool source_state source_state_pool;
bind source_state_pool *;

// initialize source
action init_source {
  input  source_state in_state;
```

```
  output source_state out_state;

  constraint in_state.initial == true;
  // fix initial values (to cut nondeterminism)
  constraint out_state.sstate == idle;
  constraint out_state.sec    == secure;
  constraint out_state.priv   == privileged;
  constraint out_state.data   == data1;
}

// source read request
action s_request_read {
  input  source_state        in_state;
  output source_state        out_state;
  output request_read_stream out_stream;

  constraint in_state.initial == false;
  // idle -¿ read
  constraint in_state.sstate  == idle;
  constraint out_state.sstate == read;
  // maintain fields
  constraint out_state.sec   == in_state.sec;
  constraint out_state.priv  == in_state.priv;
  constraint out_state.data  == in_state.data;
  // write to stream
  constraint out_stream.sec  == in_state.sec;
  constraint out_stream.priv == in_state.priv;
}

action s_grant_read {
  input  source_state      in_state;
  input  grant_read_stream in_stream;
  output source_state      out_state;

  constraint in_state.initial == false;
  // read -¿ idle
  constraint in_state.sstate  == read;
  constraint out_state.sstate == idle;
  // maintain fields
  constraint out_state.sec   == in_state.sec;
  constraint out_state.priv  == in_state.priv;
  constraint out_state.data  == in_state.data;
  // no constraint on the (thus, random) data read from the input stream
}

action s_reject_read {
  input  source_state       in_state;
  input  reject_read_stream in_stream;
  output source_state       out_state;

  constraint in_state.initial == false;
  // read -¿ idle
```

```
  constraint in_state.sstate  == read;
  constraint out_state.sstate == idle;
  // maintain fields
  constraint out_state.sec    == in_state.sec;
  constraint out_state.priv   == in_state.priv;
  constraint out_state.data   == in_state.data;
}

// source write request
action s_request_write {
  input  source_state          in_state;
  output source_state          out_state;
  output request_write_stream  out_stream;

  constraint in_state.initial == false;
  // idle -¿ write
  constraint in_state.sstate  == idle;
  constraint out_state.sstate == write;
  // maintain fields
  constraint out_state.sec    == in_state.sec;
  constraint out_state.priv   == in_state.priv;
  constraint out_state.data   == in_state.data;
  // write to stream
  constraint out_stream.data == in_state.data;
  constraint out_stream.sec  == in_state.sec;
  constraint out_stream.priv == in_state.priv;
}

action s_grant_write {
  input  source_state in_state;
  input  grant_write_stream in_stream;
  output source_state out_state;

  constraint in_state.initial == false;
  // write -¿ idle
  constraint in_state.sstate  == write;
  constraint out_state.sstate == idle;
  // maintain fields
  constraint out_state.sec    == in_state.sec;
  constraint out_state.priv   == in_state.priv;
  constraint out_state.data   == in_state.data;
}

action s_reject_write {
  input  source_state         in_state;
  input  reject_write_stream  in_stream;
  output source_state         out_state;

  constraint in_state.initial == false;
  // write -¿ idle
  constraint in_state.sstate  == write;
  constraint out_state.sstate == idle;
```

```
    // maintain fields
    constraint out_state.sec    == in_state.sec;
    constraint out_state.priv   == in_state.priv;
    constraint out_state.data   == in_state.data;
  }

  // source protection change request
  action s_request_protection {
    input  source_state               in_state;
    output source_state               out_state;
    output request_protection_stream  out_stream;

    constraint in_state.initial == false;
    // idle -¿ change
    constraint in_state.sstate  == idle;
    constraint out_state.sstate == change;
    // maintain fields
    constraint out_state.sec    == in_state.sec;
    constraint out_state.priv   == in_state.priv;
    constraint out_state.data   == in_state.data;
    // write to stream
    // no constraint on the new security and new privilege of the target
    // but source still states its security and privilege
    constraint out_stream.sec  == in_state.sec;
    constraint out_stream.priv == in_state.priv;
  }

  action s_grant_protection {
    input  source_state             in_state;
    input  grant_protection_stream  in_stream;
    output source_state             out_state;

    constraint in_state.initial == false;
    // change -¿ idle
    constraint in_state.sstate  == change;
    constraint out_state.sstate == idle;
    // maintain fields
    constraint out_state.sec    == in_state.sec;
    constraint out_state.priv   == in_state.priv;
    constraint out_state.data   == in_state.data;
  }

  action s_reject_protection {
    input  source_state              in_state;
    input  reject_protection_stream  in_stream;
    output source_state              out_state;

    constraint in_state.initial == false;
    // change -¿ idle
    constraint in_state.sstate  == change;
    constraint out_state.sstate == idle;
    // maintain fields
```

```
  constraint out_state.sec    == in_state.sec;
  constraint out_state.priv   == in_state.priv;
  constraint out_state.data   == in_state.data;
}

// change application running on the source: modify security, privilege, and data
action change_source_config {
  input  source_state in_state;
  output source_state out_state;

  constraint in_state.initial == false;
  // stay in idle
  constraint in_state.sstate  == idle;
  constraint out_state.sstate == idle;
  // no constraint: randomly change source security, privilege, and data
  // (change application running on the source)
}

// ---------------------------------------
// Finite State Machine for the Target
// ---------------------------------------

enum target_state_e {
  idle, read, write, change
}

state target_state { // Target FSM
  rand target_state_e sstate; // FSM STATE
  // Target internal data
  rand data_e data;           // current data
  rand security_e sec;        // current sec protection
  rand privilege_e priv;      // current priv protection
  // Remember last transaction
  rand security_e tx_sec;     // transaction sec
  rand privilege_e tx_priv;   // transaction priv
  rand data_e tx_data;        // transaction data (write request)
  rand security_e next_sec;   // transaction change sec request
  rand privilege_e next_priv; // transaction change priv request
}
pool target_state target_state_pool;
bind target_state_pool *;

action init_target {
  input  target_state in_state;
  output target_state out_state;

  constraint in_state.initial == true;
  // Cut nondeterminism by assigning inital values
  constraint out_state.sstate    == idle;
  constraint out_state.data      == data1;
  constraint out_state.sec       == non_secure;
  constraint out_state.priv      == non_privileged;
```

```
  constraint out_state.tx_sec    == non_secure;
  constraint out_state.tx_priv   == non_privileged;
  constraint out_state.tx_data   == data1;
  constraint out_state.next_sec  == non_secure;
  constraint out_state.next_priv == non_privileged;
}

// target READ request
action t_request_read {
  input  target_state in_state;
  input  request_read_stream in_stream;
  output target_state out_state;

  constraint in_state.initial == false;
  // Idle -¿ Read
  constraint in_state.sstate  == idle;
  constraint out_state.sstate == read;
  // save stream data
  constraint out_state.tx_sec       == in_stream.sec;
  constraint out_state.tx_priv      == in_stream.priv;
  // Maintain fields
  constraint out_state.data      == in_state.data;
  constraint out_state.sec       == in_state.sec;
  constraint out_state.priv      == in_state.priv;
  constraint out_state.tx_data   == in_state.tx_data;
  constraint out_state.next_sec  == in_state.next_sec;
  constraint out_state.next_priv == in_state.next_priv;

}

action t_grant_read {
  input  target_state in_state;
  output target_state out_state;
  output grant_read_stream out_stream;

  constraint in_state.initial == false;
  // Read -¿ Idle
  constraint in_state.sstate  == read;
  constraint out_state.sstate == idle;
  // Maintain fields
  constraint out_state.data      == in_state.data;
  constraint out_state.sec       == in_state.sec;
  constraint out_state.priv      == in_state.priv;
  constraint out_state.tx_sec    == in_state.tx_sec;
  constraint out_state.tx_priv   == in_state.tx_priv;
  constraint out_state.tx_data   == in_state.tx_data;
  constraint out_state.next_sec  == in_state.next_sec;
  constraint out_state.next_priv == in_state.next_priv;
  // Check protection
  constraint (in_state.tx_sec == secure) ||
             (in_state.sec == non_secure);
  constraint (in_state.tx_priv == privileged) ||
```

```
            (in_state.priv == non_privileged);
    // Write on stream (give the data)
    constraint out_stream.data == in_state.data;
  }

  action t_reject_read {
    input  target_state in_state;
    output target_state out_state;
    output reject_read_stream out_stream;

    constraint in_state.initial == false;
    // Read -¿ Idle
    constraint in_state.sstate  == read;
    constraint out_state.sstate == idle;
    // Maintain fields
    constraint out_state.data      == in_state.data;
    constraint out_state.sec       == in_state.sec;
    constraint out_state.priv      == in_state.priv;
    constraint out_state.tx_sec    == in_state.tx_sec;
    constraint out_state.tx_priv   == in_state.tx_priv;
    constraint out_state.tx_data   == in_state.tx_data;
    constraint out_state.next_sec  == in_state.next_sec;
    constraint out_state.next_priv == in_state.next_priv;
    // Check protection
    constraint (in_state.sec == secure) || (in_state.priv == privileged);
    constraint (
                // sec check
                ((in_state.tx_sec  == non_secure) &&
                 (in_state.sec == secure))
                ||
                // priv check
                ((in_state.tx_priv == non_privileged) &&
                 (in_state.priv == privileged))
                );
    // Write on stream (fail verdict)
  }

  // target WRITE request
  action t_request_write {
    input  target_state in_state;
    input  request_write_stream in_stream;
    output target_state out_state;

    constraint in_state.initial == false;
    // Idle -¿ Write
    constraint in_state.sstate  == idle;
    constraint out_state.sstate == write;
    // save stream data
    constraint out_state.tx_sec       == in_stream.sec;
    constraint out_state.tx_priv      == in_stream.priv;
    constraint out_state.tx_data      == in_stream.data;
    // Maintain fields
```

```
  constraint out_state.data      == in_state.data;
  constraint out_state.sec       == in_state.sec;
  constraint out_state.priv      == in_state.priv;
  constraint out_state.next_sec  == in_state.next_sec;
  constraint out_state.next_priv == in_state.next_priv;
}

action t_grant_write {
  input  target_state in_state;
  output target_state out_state;
  output grant_write_stream out_stream;

  constraint in_state.initial == false;
  // write -¿ idle
  constraint in_state.sstate  == write;
  constraint out_state.sstate == idle;
  // Check protection
  constraint (in_state.tx_sec == secure) ||
             (in_state.sec == non_secure);
  constraint (in_state.tx_priv == privileged) ||
             (in_state.priv == non_privileged);
  // update data
  constraint out_state.data == in_state.tx_data;
  // maintain fields
  constraint out_state.sec       == in_state.sec;
  constraint out_state.priv      == in_state.priv;
  constraint out_state.tx_sec    == in_state.tx_sec;
  constraint out_state.tx_priv   == in_state.tx_priv;
  constraint out_state.tx_data   == in_state.tx_data;
  constraint out_state.next_sec  == in_state.next_sec;
  constraint out_state.next_priv == in_state.next_priv;
}

action t_reject_write {
  input  target_state in_state;
  output target_state out_state;
  output reject_write_stream out_stream;

  constraint in_state.initial == false;
  // Write -¿ Idle
  constraint in_state.sstate  == write;
  constraint out_state.sstate == idle;
  // Maintain fields
  constraint out_state.data      == in_state.data;
  constraint out_state.sec       == in_state.sec;
  constraint out_state.priv      == in_state.priv;
  constraint out_state.tx_sec    == in_state.tx_sec;
  constraint out_state.tx_priv   == in_state.tx_priv;
  constraint out_state.tx_data   == in_state.tx_data;
  constraint out_state.next_sec  == in_state.next_sec;
  constraint out_state.next_priv == in_state.next_priv;
  // Check protection
```

```
  constraint (in_state.sec == secure) || (in_state.priv == privileged);
  constraint (
              // sec check
              ((in_state.tx_sec  == non_secure) &&
               (in_state.sec == secure))
              ||
              // priv check
              ((in_state.tx_priv == non_privileged) &&
               (in_state.priv == privileged))
              );
  // Write on stream (fail verdict)
}

// target Protection change request
action t_request_protection {
  input  target_state in_state;
  input  request_protection_stream in_stream;
  output target_state out_state;

  constraint in_state.initial == false;
  // idle -¿ change
  constraint in_state.sstate  == idle;
  constraint out_state.sstate == change;
  // save stream data
  constraint out_state.tx_sec    == in_stream.sec;
  constraint out_state.tx_priv   == in_stream.priv;
  constraint out_state.next_sec  == in_stream.next_sec;
  constraint out_state.next_priv == in_stream.next_priv;
  // maintain fields
  constraint out_state.data      == in_state.data;
  constraint out_state.sec       == in_state.sec;
  constraint out_state.priv      == in_state.priv;
  constraint out_state.tx_data   == in_state.tx_data;
}

action t_grant_protection {
  input  target_state in_state;
  output target_state out_state;
  output grant_protection_stream out_stream;

  constraint in_state.initial == false;
  // Change protection -¿ Idle
  constraint in_state.sstate  == change;
  constraint out_state.sstate == idle;
  // Update protection
  constraint out_state.sec  == in_state.tx_sec;
  constraint out_state.priv == in_state.tx_priv;
  // Maintain fields
  constraint out_state.data      == in_state.data;
  constraint out_state.tx_sec    == in_state.tx_sec;
  constraint out_state.tx_priv   == in_state.tx_priv;
  constraint out_state.tx_data   == in_state.tx_data;
```

```
    constraint out_state.next_sec  == in_state.next_sec;
    constraint out_state.next_priv == in_state.next_priv;
    // Check protection
    constraint (in_state.tx_sec == secure);
    constraint (in_state.tx_priv == privileged);
  }

  action t_reject_protection {
    input  target_state in_state;
    output target_state out_state;
    output reject_protection_stream out_stream;

    constraint in_state.initial == false;
    // Change protection -¿ Idle
    constraint in_state.sstate  == change;
    constraint out_state.sstate == idle;
    // Maintain fields
    constraint out_state.data      == in_state.data;
    constraint out_state.sec       == in_state.sec;
    constraint out_state.priv      == in_state.priv;
    constraint out_state.tx_sec    == in_state.tx_sec;
    constraint out_state.tx_priv   == in_state.tx_priv;
    constraint out_state.tx_data   == in_state.tx_data;
    constraint out_state.next_sec  == in_state.next_sec;
    constraint out_state.next_priv == in_state.next_priv;
    // Check protection
    constraint (
}
```

C Monolithic PSS Model

This PSS model also includes the four verification intents mentioned in Section 3.

```
component pss_top {
  //-------------------------------------
  // Types
  //-------------------------------------

  enum data_e {
    data1, data2
  }

  enum security_e {
    secure, non_secure
  }

  enum privilege_e {
    privileged, non_privileged
  }

  //-------------------------------------
```

```
// Finite State Machine for the System
// ------------------------------------

enum system_state_e {
  idle, read, write, change
}

state system_state {
  // State of the FSM encoding the SoC
  rand system_state_e sstate;    // FSM STATE

  // Information about the source IP
  rand security_e  source_sec;   // current source security
  rand privilege_e source_priv;  // current source privilege
  rand data_e      source_data;  // current source used by source for WRITE

  // Information about the target IP
  rand security_e  target_sec;   // current target security
  rand privilege_e target_priv;  // current target privilege
  rand data_e      target_data;  // current data stored in target

  // New security and privilege (only meaningful for transaction PROTECTION)
  rand security_e  new_sec;      // new target security
  rand privilege_e new_priv;     // new target privilege
}
pool system_state system_state_pool;
bind system_state_pool *;

// ------------------------------------
// Finite State Machine Actions
// ------------------------------------

// Force an initial state
action init_system {
  input  system_state in_state;
  output system_state out_state;

  // Execute only in the initial state
  constraint in_state.initial      == true;
  // Cut nondeterminism by assigning inital values
  constraint out_state.sstate      == idle;
  // Source (highst security and privilege levels)
  constraint out_state.source_sec  == secure;
  constraint out_state.source_priv == privileged;
  constraint out_state.source_data == data1;
  // Target (lowest security and privilege levels)
  constraint out_state.target_sec  == non_secure;
  constraint out_state.target_priv == non_privileged;
  constraint out_state.target_data == data1;
  // New target protection
  constraint out_state.new_sec     == non_secure;
  constraint out_state.new_priv    == non_privileged;
```

```
}

// ------------------------------------
// When in IDLE state, let the source change it's data and protection.
// This represents the change of the application currently running on the source.
action change_source_config {
  input  system_state in_state;
  output system_state out_state;

  // Do not execute in the initial state
  constraint in_state.initial == false;
  // Stay in idle
  constraint in_state.sstate  == idle;
  constraint out_state.sstate == idle;
  // Maintain target fields
  constraint out_state.target_sec  == in_state.target_sec;
  constraint out_state.target_priv == in_state.target_priv;
  constraint out_state.target_data == in_state.target_data;
  constraint out_state.new_sec     == in_state.new_sec;
  constraint out_state.new_priv    == in_state.new_priv;
  // Randomly change source security, privilege, and data
  // (change application running on the source)
}

// ------------------------------------
// READ

action s_request_read {
  input  system_state in_state;
  output system_state out_state;

  // Do not execute in the initial state
  constraint in_state.initial == false;
  // Move from Idle to Read
  constraint in_state.sstate  == idle;
  constraint out_state.sstate == read;
  // Maintain source fields
  constraint out_state.source_sec  == in_state.source_sec;
  constraint out_state.source_priv == in_state.source_priv;
  constraint out_state.source_data == in_state.source_data;
  // Maintain target fields
  constraint out_state.target_sec  == in_state.target_sec;
  constraint out_state.target_priv == in_state.target_priv;
  constraint out_state.target_data == in_state.target_data;
  constraint out_state.new_sec     == in_state.new_sec;
  constraint out_state.new_priv    == in_state.new_priv;
}

action t_grant_read {
  input  system_state in_state;
  output system_state out_state;
```

```
  constraint in_state.initial == false;
  // Move from Read to Idle
  constraint in_state.sstate  == read;
  constraint out_state.sstate == idle;
  // Check protection
  constraint (in_state.source_sec == secure) ||
             (in_state.target_sec == non_secure);
  constraint (in_state.source_priv == privileged) ||
             (in_state.target_priv == non_privileged);
  // Maintain source fields
  constraint out_state.source_sec  == in_state.source_sec;
  constraint out_state.source_priv == in_state.source_priv;
  constraint out_state.source_data == in_state.source_data;
  // Maintain target fields
  constraint out_state.target_sec  == in_state.target_sec;
  constraint out_state.target_priv == in_state.target_priv;
  constraint out_state.target_data == in_state.target_data;
  constraint out_state.new_sec     == in_state.new_sec;
  constraint out_state.new_priv    == in_state.new_priv;
}

action t_reject_read {
  input  system_state in_state;
  output system_state out_state;

  constraint in_state.initial == false;
  // Move from Read to Idle
  constraint in_state.sstate  == read;
  constraint out_state.sstate == idle;

  // Check protection
  constraint (in_state.target_sec == secure) ||
             (in_state.target_priv == privileged);
  constraint (    // security check
                 ((in_state.source_sec == non_secure) &&
                 (in_state.target_sec == secure))
               || // privilege check
                 ((in_state.source_priv == non_privileged) &&
                 (in_state.target_priv == privileged)));

  // Maintain source fields
  constraint out_state.source_sec  == in_state.source_sec;
  constraint out_state.source_priv == in_state.source_priv;
  constraint out_state.source_data == in_state.source_data;
  // Maintain target fields
  constraint out_state.target_sec  == in_state.target_sec;
  constraint out_state.target_priv == in_state.target_priv;
  constraint out_state.target_data == in_state.target_data;
  constraint out_state.new_sec     == in_state.new_sec;
  constraint out_state.new_priv    == in_state.new_priv;
}
```

```
// ------------------------------------
// WRITE

action s_request_write {
  input  system_state in_state;
  output system_state out_state;

  constraint in_state.initial == false;
  // Idle -¿ Write
  constraint in_state.sstate  == idle;
  constraint out_state.sstate == write;
  // Maintain source fields
  constraint out_state.source_sec  == in_state.source_sec;
  constraint out_state.source_priv == in_state.source_priv;
  constraint out_state.source_data == in_state.source_data;
  // Maintain target fields
  constraint out_state.target_sec  == in_state.target_sec;
  constraint out_state.target_priv == in_state.target_priv;
  constraint out_state.target_data == in_state.target_data;
  constraint out_state.new_sec     == in_state.new_sec;
  constraint out_state.new_priv    == in_state.new_priv;
}

action t_grant_write {
  input  system_state in_state;
  output system_state out_state;

  constraint in_state.initial == false;
  // Move from Write to Idle
  constraint in_state.sstate  == write;
  constraint out_state.sstate == idle;
  // Check protection
  constraint (in_state.source_sec == secure) ||
             (in_state.target_sec == non_secure);
  constraint (in_state.source_priv == privileged) ||
             (in_state.target_priv == non_privileged);
  // update data
  constraint out_state.target_data == in_state.source_data;
  // Maintain source fields
  constraint out_state.source_sec  == in_state.source_sec;
  constraint out_state.source_priv == in_state.source_priv;
  constraint out_state.source_data == in_state.source_data;
  // Maintain target fields
  constraint out_state.target_sec  == in_state.target_sec;
  constraint out_state.target_priv == in_state.target_priv;
  constraint out_state.new_sec     == in_state.new_sec;
  constraint out_state.new_priv    == in_state.new_priv;
}

action t_reject_write {
  input  system_state in_state;
  output system_state out_state;
```

```
  constraint in_state.initial == false;
  // Write -¿ Idle
  constraint in_state.sstate  == write;
  constraint out_state.sstate == idle;
  // Check protection
  constraint (in_state.target_sec == secure) ||
             (in_state.target_priv == privileged);
  constraint (    // security check
                  ((in_state.source_sec == non_secure) &&
                   (in_state.target_sec == secure))
               || // privilege check
                  ((in_state.source_priv == non_privileged) &&
                   (in_state.target_priv == privileged)));
  // Maintain source fields
  constraint out_state.source_sec  == in_state.source_sec;
  constraint out_state.source_priv == in_state.source_priv;
  constraint out_state.source_data == in_state.source_data;
  // Maintain target fields
  constraint out_state.target_sec  == in_state.target_sec;
  constraint out_state.target_priv == in_state.target_priv;
  constraint out_state.target_data == in_state.target_data;
  constraint out_state.new_sec     == in_state.new_sec;
  constraint out_state.new_priv    == in_state.new_priv;
}

// ------------------------------------
// Change PROTECTION of target

action s_request_protection {
  input  system_state in_state;
  output system_state out_state;

  constraint in_state.initial == false;
  // Move from Idle to Change
  constraint in_state.sstate  == idle;
  constraint out_state.sstate == change;
  // Maintain source fields
  constraint out_state.source_sec  == in_state.source_sec;
  constraint out_state.source_priv == in_state.source_priv;
  constraint out_state.source_data == in_state.source_data;
  // Maintain target fields
  constraint out_state.target_sec  == in_state.target_sec;
  constraint out_state.target_priv == in_state.target_priv;
  constraint out_state.target_data == in_state.target_data;
  // Randomly select new target security and privilege
}

action t_grant_protection {
  input  system_state in_state;
  output system_state out_state;
```

```
  constraint in_state.initial == false;
  // Move from Change to Idle
  constraint in_state.sstate  == change;
  constraint out_state.sstate == idle;
  // Check protection
  constraint (in_state.source_sec == secure);
  constraint (in_state.source_priv == privileged);
  // Update target protection
  constraint out_state.target_sec  == in_state.new_sec;
  constraint out_state.target_priv == in_state.new_priv;
  // Reset new target protection
  constraint out_state.new_sec     == non_secure;
  constraint out_state.new_priv    == non_privileged;
  // Maintain source fields
  constraint out_state.source_sec  == in_state.source_sec;
  constraint out_state.source_priv == in_state.source_priv;
  constraint out_state.source_data == in_state.source_data;
  // Maintain target fields
  constraint out_state.target_data == in_state.target_data;
}

action t_reject_protection {
  input  system_state in_state;
  output system_state out_state;

  constraint in_state.initial == false;
  // Move from Change to Idle
  constraint in_state.sstate  == change;
  constraint out_state.sstate == idle;
  // Check protection
  constraint (// security check
              (in_state.source_sec  == non_secure) ||
              // privilege check
              (in_state.source_priv == non_privileged));
  // Reset new target protection
  constraint out_state.new_sec     == non_secure;
  constraint out_state.new_priv    == non_privileged;
  // Maintain source fields
  constraint out_state.source_sec  == in_state.source_sec;
  constraint out_state.source_priv == in_state.source_priv;
  constraint out_state.source_data == in_state.source_data;
  // Maintain target fields
  constraint out_state.target_sec  == in_state.target_sec;
  constraint out_state.target_priv == in_state.target_priv;
  constraint out_state.target_data == in_state.target_data;
}

// ------------------------------------
// Verification intents
// ------------------------------------

// Any reject
```

```
action intent_1 {
  t_reject_read       Reject_Read;
  t_reject_write      Reject_Write;
  t_reject_protection Reject_Protection;
  activity {
    select{
      Reject_Read;
      Reject_Write;
      Reject_Protection;
    }
  }
}

// All responses (interleaving semantics)
action intent_2 {
  t_grant_read        Grant_Read;
  t_grant_write       Grant_Write;
  t_grant_protection  Grant_Protection;
  t_reject_read       Reject_Read;
  t_reject_write      Reject_Write;
  t_reject_protection Reject_Protection;
  activity {
    schedule{
      Grant_Read;
      Grant_Write;
      Grant_Protection;
      Reject_Read;
      Reject_Write;
      Reject_Protection;
    }
  }
}

// All responses (sequential semantics)
action intent_3 {
  t_grant_read        Grant_Read;
  t_grant_write       Grant_Write;
  t_grant_protection  Grant_Protection;
  t_reject_read       Reject_Read;
  t_reject_write      Reject_Write;
  t_reject_protection Reject_Protection;
  activity {
    Grant_Read;
    Grant_Write;
    Grant_Protection;
    Reject_Read;
    Reject_Write;
    Reject_Protection;
  }
}

// Access data with different security and/or privilege
```

```
  // This test scenario executes the following steps:
  // 1) Elevate target security/privilege
  // 2) Write data to target with same security/privilege as the target
  // 3) Change source security/privilege, keeping target security/privilege unchanged
  // 4) read the target
  // We did not find how to express unwanted behavior (e.g., how to forbid to change target security/privilege).
  // Thus we request that there is a change of the source configuration immediately after the write transaction
  // was granted and rely on the shortest path to avoid any further change of the target security/privilege before
  // the read transaction.
  action intent_4 {
    change_source_config Change_Source;
    t_grant_read         Grant_Read;
    t_grant_protection   Grant_Protection;
    t_grant_write        Grant_Write;
    activity {
      Grant_Protection;   // get s and p
      Grant_Write;        // do an accepted write with the same s and p
      Change_Source;
      // Rely on shortest path to not do any other Grant_Protection
      Grant_Read;   // do an accepted read with another s or another p
    }

    constraint {
      // Grant_Write with the same security and privilege as Grant_Protection
      Grant_Write.in_state.source_sec ==
        Grant_Protection.out_state.source_sec;
      Grant_Write.in_state.source_priv ==
        Grant_Protection.out_state.source_priv;

      // Read granted to a source different security or privilege as the Write
      ((Grant_Read.in_state.sec  != Grant_Write.out_state.sec) ||
       (Grant_Read.in_state.priv != Grant_Write.out_state.priv));

      // Read with same target security and privilege as Grant_Protection
      // (no guarantee of absence of change in between)
      Grant_Read.in_state.target_sec ==
        Grant_Protection.out_state.target_sec;
      Grant_Read.in_state.target_priv ==
        Grant_Protection.out_state.target_priv;

      // Read the data that was written
      // (no guarantee of absence of change in between)
      Grant_Read.in_state.target_data == Grant_Write.out_state.target_data;
    }

    // Allow nothing between Grant_Write and Change_Source
    // This prevents all other actions because there is only one flow object
    bind Grant_Write.out_state Change_Source.in_state;
  }
}
```

D SVL Script for all Verification Steps

The following SVL script[11] generates and compares the LTSs mentioned in Sections 2.1 and 2.2. It requires the translation to LNT of the monolithic PSS model (available in the MARS model repository).

```
-- generation of the LTS for a SoC with 8 source IPs

"model_8_1.bcg" =
   reduction of "model_8_1.lnt";

-- generation of the LTS for a SoC with a single source IP

"model_2_1.bcg" =
   reduction of
   -- remove all actions from absent sources (only IP1 is present)
   total cut all but "[^!]*-!IP1-.*" in
   "model_8_1.lnt":"SOC_2";

-- generation of the LTS for the PSS model

"RI_monolithic.bcg" =
   strong reduction of
   weak trace reduction of
   branching reduction of
   -- 4. suppression of supperfluous offers (to be completed)
   total rename
      "\(CHANGE_SOURCE_CONFIG\)-.*-\(![^!]*-![^!]*-![^!]*\)-![^!]*-![^!]*-!
         [^!]*-![^!]*-![^!]*" -> "\1-\2",
      "\(GRANT_READ\)-.*-\(![^!]*\)-![^!]*-![^!]*" -> "\1-\2",
      "\(GRANT_WRITE\).*" -> "\1",
      "\(GRANT_PROTECTION\)-.*-\(![^!]*-![^!]*\)-![^!]*-![^!]*-![^!]*" -> "
         \1-\2",
      "\(REJECT_[A-Z]*\).*" -> "\1",
      "REQUEST_\(READ\)-![^!]*-\(![^!]*-![^!]*\)-.*" -> "\1-\2",
      "REQUEST_\(WRITE\)-![^!]*-\(![^!]*-![^!]*-![^!]*\)-.*" -> "\1-\2",
      "REQUEST_\(PROTECTION\)-![^!]*-\(![^!]*-![^!]*\)-.*-\(![^!]*-![^!]*\)
         " -> "\1-\2-\3"
   in
   -- 3. suppression of the prefix SOURCE/TARGET
   rename
      "SOURCE_\([A-Z_]*\)" -> "\1",
      "TARGET_\([A-Z_]*\)" -> "\1"
   in
   -- 2. removal of the first offer (indicating the action)
   total rename
      "\([^!]*\)!PSS_TOP_X_[^!]*\(.*\)" -> "\1\2"
   in
   -- 1. removal of the gate prefix "PSS_TOP_X_"
   rename
      "PSS_TOP_X_\(.*\)" -> "\1"
```

[11] SVL (Script Verification Language) is the language for describing verification scenarios for the CADP toolbox.

```
    in
    -- hiding initialisation and interaction with the state flow object
    divbranching reduction of
    hide
        ".*OUTPUT",
        ".*INPUT",
        ".*INIT_SYSTEM"
    in
        "../PSS/RI_monolithic.lnt"
    end hide;

-- comparion of the three LTSs

property MODEL_EQUIVALENCE
    "after-hiding-ip-identities,-alls-models-are-equivalent"
is
    branching comparison
        hide CHANGE_SOURCE_CONFIG in
        total rename "\([^-]*\)-![^!]*-![^!]*\(.*\)" -> "\1-\2" in
        "model_8_1.bcg"
        ==
        hide CHANGE_SOURCE_CONFIG in
        total rename "\([^-]*\)-![^!]*-![^!]*\(.*\)" -> "\1-\2" in
        "model_2_1.bcg";
    expected TRUE;

    branching comparison
        hide CHANGE_SOURCE_CONFIG in
        total rename "\([^-]*\)-![^!]*-![^!]*\(.*\)" -> "\1-\2" in
        "model_8_1.bcg"
        ==
        hide CHANGE_SOURCE_CONFIG in
        "RI_monolithic.bcg";
    expected TRUE;
end property
```