PSL: Beyond Hardware Verification

Ziv Glazberg, Mark Moulin, Avigail Orni, Sitvanit Ruah, and Emmanuel Zarpas

Abstract In recent years, the language PSL (Property Specification Language, a.k.a. IEEE P1850) has been embraced and put to successful use by chip design/verification engineers across the electronics industry. While PSL is mainly used for hardware verification, it can, in fact, be used to verify a wide variety of systems, including missile interception systems, railway interlocking protocols, system automation policies, and even business processes. We discuss and exemplify how PSL can be used as a general purpose language for the specification of models and properties, beyond hardware systems.

Keywords: PSL, nonlinear controllers, concurrent reactive systems, policy-based system automation.

1 Introduction

Since its approval by the IEEE, the Property Specification Language (PSL IEEE P1850, [21]) has met with huge success in the hardware verification community. It is widely used for industrial hardware verification and is supported by a wide range of vendors. PSL is mainly considered a hardware specification language; however, its use is not restricted to hardware verification. While some features such as clocking are close to hardware, on the whole PSL is a general property specification language.

The goal of this paper is to illustrate how PSL can be used outside the hardware verification scope. Section 2 describes PSL. Section 3 focuses on the use of PSL for the modeling and verification of antimissile interception hybrid control system. Section 4 describes our work in statically checking the output of IBM Rational Rose Real-Time (RoseRT), a widely used model-driven development tool for concurrent reactive systems. Section 5 deals with the modeling and verification of Tivoli System Automation policies.

2 Property Specification Language (PSL)

PSL, the Property Specification Language, is a language for specifying properties. It is typically used for specifying temporal properties of systems, i.e., properties that deal with the behavior of a system over time. The main part of PSL is based on the temporal logic Linear Time Logic (LTL) [13], augmented with regular expressions. PSL originated as the Sugar language, which was used by the IBM RuleBase PE model checking tool [22], and later evolved into an IEEE standard.

This section presents a brief overview of PSL. Only a small selection of the language is shown here. A clear and comprehensive introduction to PSL can be found in [7]. The official definition of PSL is in IEEE Std. 1850-2005 [21].

2.1 Simple PSL Examples

We consider a system that accepts requests of some sort and processes them. The assumption is that our system has some definition of time points, which may be points at which a system clock ticks (if the system is synchronous), or points at which certain chosen events occur. PSL only requires that we have a sequence (finite or infinite) of discrete time points. (The notion of time in PSL is discussed more fully in the "The Granularity of Time" section.)

Our system has variables such as `req`, `ack`, `start`, `busy`, and `done`. Each variable is true at certain time points. We demonstrate how each of the following English statements, which describe system behavior, can be formulated in PSL.

- "Whenever `start` is true at a time point, `busy` will be true at the following time point."

  `always (start -> next busy)` (1)

- "For every occurrence of `req` that is immediately followed by `ack`, processing of the acknowledged request begins at the next time point after the `ack`. The processing sequence begins with `start`, which is followed by `busy` for some number of time points, and ends with `done`."

  `{[*]; req; ack} |=> {start; busy[*]; done}` (2)

PSL is mathematically rigorous, therefore the properties in PSL are precise and unambiguous. However, they are also easy to read. Thus, a specification written in PSL can be used as input for automatic tools and may also serve as part of a human-readable specification document. In the following section, we present some PSL constructs and operators. Many of the PSL operators are based on LTL operators. Other PSL constructs are based on SEREs (discussed below), which are a type of regular expression. Another set of PSL operators is based on the CTL language [9] and is not discussed here.

2.2 SEREs – Regular Expressions in PSL

PSL includes a type of regular expression called a *SERE*, a *Sequential Extended Regular Expression*. SEREs are used to describe scenarios. The simplest type of SERE is a sequence of Boolean expressions separated by semicolons, such as {req; !ack; ack}. This SERE describes a scenario spanning three time points, in which req holds at the first time point, ack does not hold at the second, and ack does hold at the third.

Generally, a SERE may describe a *set* of scenarios. For example, the operator [*] indicates an interval of zero or more time points, in which anything may occur. Therefore, the SERE {start;[*];done} describes any scenario that begins with start and ends with done. The [*] operator may also be attached to a Boolean expression. The expression busy[*] describes an interval of zero or more time points in which busy is true.

Additional operators serve as shorthand for longer constructions. For example, {busy[*4]} is equivalent to {busy; busy; busy; busy}. For any constant number *n*, the expression busy[*n] describes a sequence of exactly *n* time points, where busy holds for all the points. The expression req[=n] describes *n* occurrences of req, which may be non-consecutive.

SERE conjunction and disjunction operators create compound SEREs. The conjunction of two SEREs, using the && operator, describes two scenarios that happen simultaneously.

For example, in {start; busy[*]; done} && {cancel[=1]} the left-hand side sub-SERE states that processing takes place (starting with start and ending with done). On the right-hand side, cancel occurs exactly once. In the conjunction, both sides happen simultaneously, so cancel occurs exactly once, at some time point, while processing is in progress.

The disjunction of two SEREs, using the | operator, describes a scenario in which either the left-hand side or the right-hand side of the disjunction (or both) must occur.

SERE operators may also be nested and combined, as shown:

```
{{busy[*]} && {cancel[=0]}} | {{req; ack}[*2]}
```

2.3 PSL Properties with SEREs

SEREs may be used as building blocks of PSL properties. Typically, a property may be composed of SEREs using the *suffix implication* operator |=>. For example

```
{[*]; req; ack} |=> {start; busy[*]; done}        (3)
```

This property states that any occurrence of the left-hand side scenario must be followed by an occurrence of the right-hand side scenario. In this particular case, {[*]; req; ack} describes a sequence of req followed immediately by ack, which may occur at any time point (due to the [*] at the beginning of the SERE). The property states that such a sequence must immediately be followed (starting at the next time point) by a scenario matching {start; busy[*]; done}. This property makes a requirement for any occurrence of a {req; ack} sequence, at any time point, including overlapping occurrences. The following property is very similar to Formula 3:

```
{[*]; req; ack} |=> {start ;busy[*]; done}!        (4)
```

This property uses the *strong* version of the right-hand side SERE. Generally, a SERE is *strong* if it is followed by an exclamation point (!), and *weak* if it is not. The property in Formula 3 is satisfied by a right-hand side scenario in which done never occurs, and busy stays true until the end of the scenario. The property in Formula 4 is only matched by a scenario in which done eventually occurs.

2.4 Other Property Styles

A PSL property may be written without using SEREs at all. For example:

```
always (start -> next busy)        (5)
```

The sub-property `(start -> next busy)` uses the logical implication operator ->, which has the standard "if-then" meaning. The `next` operator refers to the time point that immediately follows the current one. So `(start -> next busy)` means, "if `start` is true at the current time point, then `busy` must be true at the next time point". Applying `always` to this sub-property means that the sub-property must hold at every time point. So the entire property means, "Whenever `start` is true, `busy` must be true at the time point that immediately follows".

The `eventually!` operator can be used to state that `start` must occur at some time point after `req` occurs (or simultaneously with `req`), as follows:

```
always (req -> eventually! start)        (6)
```

In addition, we can combine non-SERE operators with SEREs, creating properties such as `(always {req; ack}|=> (start && eventually! done))`

2.5 PSL Layers and Flavors

PSL is structured in four layers: the Boolean layer, which contains the Boolean expressions used in properties; the temporal layer, which contains the temporal properties and SEREs; the verification layer, for directing the use of PSL by a tool; and the modeling layer, for modeling behavior of inputs and auxiliary variables.

Based on this layered structure, several flavors of PSL have been defined. The most generic is the GDL flavor, which is based on the General Description Language (GDL). GDL was designed especially for use in the PSL modeling layer, and can be used for modeling systems in diverse problem domains, at various levels of abstraction.

The other PSL flavors are based on hardware description languages (HDLs). In each flavor, the Boolean and modeling layers follow the syntax of the underlying HDL (or of GDL). The temporal and verification layers are not affected by flavors.

2.6 The Granularity of Time

Time in PSL is discrete; that is, time advances in pre-defined units. For example, the property `((size == 3) -> next (size > 5))` requires that if size equals 3 now, then at the next time point `(size > 5)` must hold. The time points are set by the system, that is, the system being verified has some function that advances by one time point.

Some PSL operators are not affected by the granularity of time. For example:

- `never (at_critical [1] && at_critical [2])` requires that `at_critical [1]` and `at_critical [2]` are never true at the same time.
- `eventually!(p)` requires that `p` holds now or at some point in the future.
- `next_event(p)(f)` requires that at the first time point at which `p` holds, `f` holds, regardless of the granularity of time.

The PSL operators that are affected by the granularity of time are next, next_e, next_a and some of the SERE operators.

The user can change the granularity of time given by the system using the @ operator. For example, assume you want to advance one time point whenever (status==OK). You can use the PSL @ operator on your property as follows:

```
(always ((size==3) -> next (size > 5)))@(status==OK)
```
(7)

3 Missile Interception Control System

It is a considerable challenge to verify aerospace hybrid control systems, especially when they include a significant amount of nonlinear dynamics. Recently, in [14], different nonlinear controllers were applied to a complex aircraft design problem. While each of the controllers demonstrated some ability to achieve the design criteria, it was noted that the controllers are labor intensive to design and would require new approaches for verification other than simulations. In this section, we present PSL-enhanced formal verification of antimissile control system.

An initial step in formal verification of a hybrid system is to make a reasonable approximation (discretization) of the nonlinear dynamics to reduce the possibly infinite state space system into a finite state space system. A weakness of such a conservative approximation is that it may require a large number of samples, and both the memory requirements and the computation time of a formal verification tool may

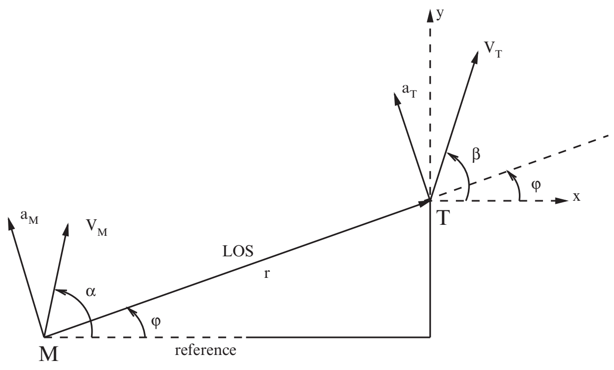

Fig. 1 Geometry of the planar tracking problem

soon become impractical [15]. Usually, the number of states to examine is huge in applications of practical interest, but in the case of missile tracking, the physical process is time limited, and can be translated into a bounded model checking application. Formal methods verify the antimissile interception using the same difference equations representation of the system used by the Matlab simulation tool, and the system properties have straightforward descriptions in PSL.

Consider the problem of a planar moving target interception, depicted in Figure 1 [10, 12]. Both the target T and pursuing missile M are assumed to be point masses moving in a plane. These are the polar system equations of motion that appear in [10]:

$$\begin{aligned} \dot{r} &= V_T \cos(\beta - \varphi) - V_M \cos(\alpha - \varphi) \\ \dot{\varphi} &= V_T \sin(\beta - \varphi) - V_M \sin(\alpha - \varphi) \end{aligned} \tag{8}$$

where LOS is the instantaneous missile-target line-of-sight, a time-variant vector from the pursuer to target; r is the range i.e., the length of the LOS; φ is the bearing angle i.e., the angle between the LOS and the reference line; α is the missile heading angle; and β is the target heading angle.

The missile is governed by its guidance system, i.e., a compensation network placed in series with engagement process Eq. (8) to accomplish an interception. Most of the applied guidance laws belong to the family of Proportional Navigation guidance laws [11], which have shown good performance against moderately-maneuvering targets. In the True Proportional Navigation guidance laws, missile acceleration is usually applied normal to LOS:

$$\begin{cases} a_M = -\lambda \dot{r} \dot{\varphi} \\ \dot{V}_M = a_M \sin(\alpha - \varphi) \\ \dot{\alpha} = a_M / V_M \cos(\alpha - \varphi) \end{cases} \tag{9}$$

where $\lambda > 0$ is a navigation constant. In practice, the control dynamic is implemented as a sampled control system (a computer-based control).

The closed-loop hybrid tracking system has a free system input; this is the target acceleration. An intelligent target is expected to perform evasive maneuvers to increase the probability of its escape. The target maneuver considered below in Eq. (10) is restricted to the application of the lateral target acceleration normal to the target velocity; this governs the following input dynamics for the target:

$$\begin{cases} a_T = b/(r\dot{\varphi}) \\ \dot{V}_T = a_T \sin(\beta - \varphi) \\ \dot{\beta} = a_T / V_T \cos(\beta - \varphi) \end{cases} \tag{10}$$

where $b > 0$ is a positive constant.

The tuning of this hybrid control system is a very tedious task. The performance of the derived guidance law is characterized by the capturability (i.e., the ability of the guidance law to ensure the capture or interception of a target), which is translated into the capture region bounds. The capture regions may not exist when the initial conditions on range, bearing angle, and their rates are high. Usually, the qualitative analysis technique is used to obtain the capture region from the chosen target maneuver, final time to intercept, and sufficient initial conditions for interception [10]. The resulting controller is relevant only to this specifically chosen target maneuver according to the specific b parameter value.

In contrast to this, formal methods provide a full coverage of the impacts of b parameter perturbations by verifying a logical model of the system to satisfy/dissatisfy the particular properties. Formal verification formulates the design problem as follows: define the particular final time and initial conditions, and find the guidance system parameters that could prevent the escape of the target under these conditions. This property is realistic and helps design robust controllers, which take into account realistic target maneuvers.

Systems that have been traditionally analyzed by formal methods are discrete; therefore, the continuous-time Eq. (8) is transformed into the difference equation presentation of the continuous-time systems as a periodically-updated system. The overall system is described in the Verilog language, which has a powerful compiler that can synthesize the Verilog model into the logical circuit of basic logical gates. All numbers are represented by 32-bit vectors.

The tracking algorithm is applied to a realistic interception scenario with the initial range varying around 30 kilometers for a maneuvering target. For this case, the property is based on a necessary capturability condition that after 10 seconds from the start of the interception process (or after the k sequences of state transitions), the distance r between the missile and target must always decrease from the initial range of 30000 meters to less than 20000 meters. The specially-chosen gain of the controller (navigation constant) $\lambda = 3$ must ensure this. The simulation procedure usually used to check controller consistency is the launch of Monte Carlo trials. A formal verification engine found a counterexample in a single run after verifying the following property formulated in PSL:

```
Property 1:
define λ = 3;
always(range(0)=30000 − > next[k] (range(k)<20000))          (11)
```

relative to all possible perturbations of target acceleration parameter $b =$ [950… 1000] with granularity 5.

The SAT solver was launched for $k = 5$ sequences of state transitions (cycles). The counterexample provides the target acceleration that caused the range to be at least 21000 meters after the first 10 seconds of the interception process. Because the designed controller with $\lambda = 3$ has not met the requirements of the intermediate range value, we must tune the navigation constant of the guidance law Eq. (9). The following property claims that it is impossible to find the navigation constant $\lambda = [3 \ldots 5]$ with granularity 0.1 that ensures the condition $r(5) < 20000$ meters.

```
Property 2:
forall λ = [3...5]
always (range=30000 -> next[k] (range>20000))                (12)
```

After a 94 second run, the RuleBase PE provides a counterexample to Property 2, saying that navigation constant $\lambda = 3.8$ is a suitable control gain for this case.

This successful example illustrates a general methodology for formal analysis of a control system:

1. Create a discrete model of the system and control law.
2. In PSL, specify the system properties.
3. Verify the initial solution (model vs. properties).
4. If system properties hold, the system design is acceptable.
5. Otherwise, select a (possibly new) candidate control law.
6. Choose a new set of candidate values for control parameters.
7. Create a model of the system with the new control law.
8. Specify the fail claim property: the selected control law, for all perturbations of control parameters, always fails.
9. Check the model against the fail claim property.
10. If the fail claim property holds, go back to Step 5 or 6.
11. If the fail claim property does not hold, conclude that the values of the control parameters provided in the counterexample are robust.

This methodology provides a possibility of efficient application of formal analysis to design and verification of control systems.

4 SMARRT: Static Model Checking and Analysis for Rose Real-Time

IBM Rational Rose Real-Time (RoseRT) is a widely used model-driven development tool for concurrent reactive systems. The system's behavior is specified using a collection of UML [5] diagrams. RoseRT generates code based on the given model. Since RoseRT is intended to support the entire development process and not just the design stage, it allows the user to execute and debug the system at the model level.

The model defines the system in an exact and complete representation so that there is no disparity between the code and the model. In the SMARRT project, we set a goal to allow users of RoseRT to formally verify the model. As opposed to traditional testing, formal verification can prove the absence of a bug, and not only the existence of a feature. Often in the model checking process, verification expertise is required to build the model, define the specifications, and observe and understand the results. We, instead, intend to equip RoseRT users with the advantages of model checking, without requiring the user to be an expert in the field.

4.1 Defining the Model

The RoseRT model describes the entire system, therefore it can be used as the model that is checked. Using a simple transformation [8], the same behavior is expressed in a PSL model. In RoseRT, the user can work with certain building-blocks that are available for describing the behavior of the system. For example, one building block is a simple message queue. Generated code based on the RoseRT model uses an efficient code template to perform the desired behavior. The efficiency of this code should not be misunderstood—it is optimized for execution but not for verification. Thus, we have hand-modeled these building-blocks in a verification-efficient template. When the RoseRT model is translated into PSL, the PSL template is used, just as the code template is used when the model is translated to code.

Figure 2 shows a small client-server protocol example [6]. This example has two state machines: one for the client and one for the server. They are connected using

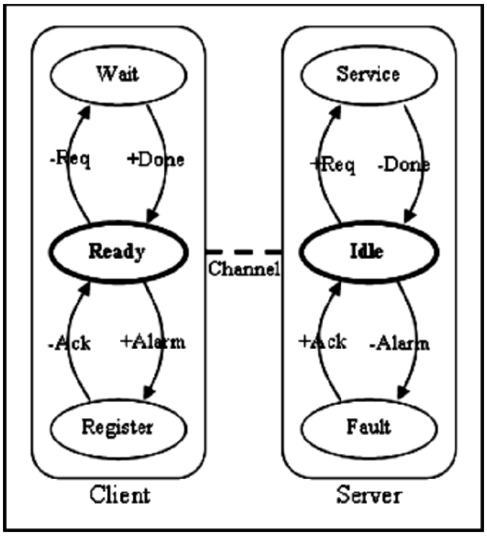

Fig. 2 Client–server protocol

a two-way message queue. The client state machine has three states: Ready, Wait, and Register. The Server state machine also has three states: Idle, Service, and Fault. A state machine changes the state, either due to a message it receives or an internal event. In this protocol, each transition is either associated with the reception or dispatch of a message. A state transition is represented by an arc. The arc is tagged by a sign: plus (+m) represents message reception and minus (−m) represents message dispatch. Initially, the client is in the Ready state and the server is in the Idle state.

SMARRT generates a PSL model that includes a state variable for each state machine defined in a state diagram, and a variable for each state machine that defines non-deterministically which port the state machine will check (the environment port, a communication port, or none). The variable that defines the state of the state machine changes according to the content of the checked port and the current state. If a transition occurs, the appropriate port action is executed (whether sending a message or getting a message). Different interleavings are modeled by allowing the state machine to not examine any port, and thus forcing it to stay in the same state.

4.2 Defining the Specification

SMARRT takes advantage of the fact that RoseRT users are comfortable working with models. The specification is written using an extension of the UML sequence diagram. Sequence diagrams depict the interactions between objects and their states. They define a clear timeline, allowing the user to express temporal specifications with a user-friendly interface. The specification is later translated into a PSL formula that needs to be verified. Even if users are not familiar with the PSL, they can express complex constraints that the verified system needs to hold. For a verification process to be successful, the user must understand which specification to verify. This is key to the usefulness of the entire process. Though the user should know which "questions" to ask, SMARRT allows the users to express these questions without prior PSL expertise.

Figure 3 shows a specification requirement over the client–server protocol specifying that the client does not enter into a deadlock state. A PSL translation of this requirement would be `eventually! Client_State=Ready`.

4.3 Model Checking the PSL Model

The translated PSL model and specification is fed to IBM RuleBase PE [4]. RuleBase PE can utilize different model checking engines (e.g., SAT-based, BDD-based, abstraction-refinement, and others) to verify the model. Though a specialized software-oriented engine exists for RuleBase PE [2, 3] we do not utilize it in verifying these models. We observe that concurrent reactive systems are more similar to hardware than common software. Typically, in software systems, a relatively small

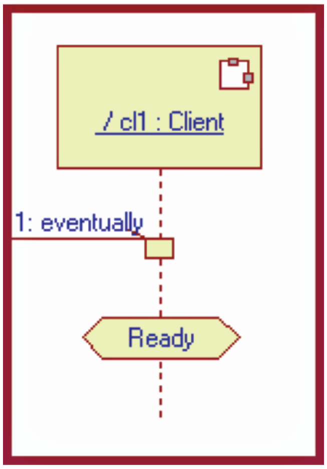

Fig. 3 Specification equivalent to "eventually! Client_State=Ready"

set of variables changes in every cycle, but in hardware, all variables may change in every cycle. Similarly, in communicating state machines for concurrent reactive systems, all machines may change their states in every cycle. Due to this similarity, we believe that hardware model checking techniques are more likely to succeed on these systems. RoseRT models sometimes introduce hierarchy into the model. Once the code is generated, the entire hierarchical structure is flattened. Though the model that needs to be verified is the flattened model, the hierarchy may be used for abstraction purposes if the model is too big for the model checker to cope with [16].

4.4 Counterexample Generation

If the model is verified and a counterexample is found, it is presented to the user in a simple and straightforward manner. The counterexample is described using a standard UML sequence diagram. As every sequence diagram depicts a possible execution of the system, the counterexample is exactly that—an execution of the system that violates the specification.

5 System Automation

System automation through policy-based management allows IT administrators to define high-level policies for various management tasks, such as networked systems and applications for business environments, network planning, problem detection, and quality of service provisions. This approach (e.g., [17]) to system management allows the separation of the rules that govern behavioral choices of the system from the functionality provided by that system. In a very general way, policies are plans of an organization to achieve its objectives. A policy can be understood as a high-level specification of the system to be automated. It is, therefore, natural to translate it into a formal language and then verify it. Here, we consider policies for the IBM Tivoli System Automation (TSA) for Multi-Platform [19]. This section focuses on how to model and check TSA policy with PSL. We translate real-life industrial policies into PSL and then verify the system with the RuleBase PE model checker.

A TSA policy is a collection of relationships that describes the automated behavior to be enforced by TSA. TSA describes temporal relationships (e.g., A should start after B) or topological relationships (e.g., A is co-located with B) between resources that should be enforced by the system. The building blocks of TSA policies are resources, which can be any piece of hardware or software in the TSA management scope, located on several nodes of the system. There are three types of resources in the TSA policy language: fixed resources (`Resource`), floating resources (`MoveGroup`), and references to a resource outside the management scope of TSA (`ResourceReference`). Resources can be grouped using the `ResourceGroup` or `Equivalency` constructs, so that they are easier to handle. TSA policies are described with XML syntax. See [20] for a detailed description of the TSA policy language.

5.1 Modeling TSA Policies with PSL

To model TSA policies, fixed and floating resources are modeled as state machines in the GDL flavor of the PSL modeling layer and the relationships are modeled as PSL assumptions. In the systems we are modeling, time is continuous, while PSL time is discrete. We deal with this by allowing events to happen at a non-deterministic time and by considering the atomic unit of time to be the minimum possible time between two events in the system.

The TSA description provides the name and node of a resource. A resource can have five states: Unknown, Online, Offline, FailedOffline, and StuckOnline. A resource state is Unknown when its state is not known by TSA for some reason; a resource is Online when it is running and Offline when it is not running. A resource is FailedOffline when it is down with a fatal failure and StuckOnline when it is running with a fatal failure. Possible transitions (where a transition takes one atomic unit of time) for the resource state:

Unknown	*-> Unknown | Online | Offline | FailedOffline | StuckOnline*
Online	*-> Unknown | Online | Offline | FailedOffline | StuckOnline*
Offline	*-> Unknown | Online | Offline | FailedOffline*
FailedOffline	*-> FailedOffline*
StuckOnline	*-> StuckOnline*

The amount of time a resource stays in a specific state is non-deterministic and independent of the behavior of other resources. Resources, resource groups, move groups, and equivalencies are coded in the PSL modeling language as an array. The first part of the array codes the node and the second part codes the state. For simplicity's sake, we denote the node and the state of a resource "r" by r.node and r.state. Resource transitions are constrained by the relationship.

Relationships are modeled as constraints using the PSL verification layer directive assume (the assume statement allows specifying an invariant). This allows us to provide a formal description of TSA policy relationships that are only informally described in [19] and [20]. We give a few examples of the way relationships are modeled in PSL.

A StartAfter B means that A must start after B starts. More precisely, when A starts, B should already be online. This translates to the following PSL verification directive:

```
assume always(rose(A.state=Online) -> (B.state=Online &
!rose(B.state=Online))) ;
```

This means the following property should be an invariant of the model: when A goes online, B should be online but did not go online at the same moment as A (*always p* is PSL syntax for the LTL *Gp*). This is more complex than expected. Translation to PSL allows a clearer and non-ambiguous description of the relationships.

A StopAfter B means that A must stop after B does, i.e., when A stops, B is already offline:

```
assume always (fell(A.state=Online) -> (B.state in
{Offline, FailedOffline} & !rose(B.state in {Offline,
FailedOffline}))) ;
```

A Collocated B means that if A is online, A and B are on the same node:

```
assume always ((A.state=Online) -> A.node=B.node) ;
```

A Anti Collocated B means that if A is online, A and B are not on the same node:

```
assume always ((A.state=Online) -> A.node!=B.node) ;
```

5.2 Verification

Once the model is built, we can check PSL properties against it to perform conflict detection, validation of the specified policy to ensure it is consistent with the capabilities of the system, deadlock detection, and loop detection.

We don't check how the system managed by TSA behaves; rather, we check properties on the policy controlling its behavior. For example, we check that the policy is not over-constrained in ways that prevent the system from running satisfactorily, we check that the system can reach the desired state, and we identify whether there exists a single point of failure with regard to these properties. The following PSL properties should hold for every policy:

```
1. assert EF nominal_state ;
2. assert AG EX true ;
3. assert AG (desired_state1 -> EF desired_state2) ;
4. assert AG (desired_state1 -> EX desired_state1) ;
```

where `nominal_state` is true when all resource groups are in the desired states specified in the policy, and `desired_state1` and `desired_state2` are chosen non-deterministically from all the desired states of the system. A desired state of the system is a state in which each resource group is in a known state, and not failed or stuck. Thus there are two "good" values per resource group: Online and Offline, and 2^n possible values of `desired_state1 and desired_state2.`

Property 1 means the system can reach the nominal state specified by the policy. Property 2 means the system can always follow the policy; i.e., there is no truncated path. Property 3 means that while running and in a desired state, the system can reach any other desired state (for instance, if resource group X is offline, all other resource groups are online, and it is possible to bring the desired resource group X online and take groups Y and Z offline). Property 4 means that once the system reaches a desirable state, it can stay there forever (this can be seen as some sort of termination property; it ensures, for instance, that no loop prevents the system from staying as long as needed in the desired state). RuleBase automatically checks that the model is not empty. These properties are rather different to the properties commonly used in hardware verification; as, for example, in [18]. The most commonly used properties for hardware verification are safety properties; non-LTL properties are uncommon. The properties we have shown so far should hold for every policy, and thus checking them can be completely automated. In addition, it is possible to perform policy-specific checks using RuleBase PE.

We built an *ad hoc* translator that semi-automatically translates the XML TSA policy into a model (described in the previous section) and extracts the definitions needed for the automated properties. We then checked these properties with the RuleBase PE model checker. Our work was used to verify several real-life TSA policies.

6 Conclusion

In this paper, we encourage readers to view PSL from a novel perspective, such that its use should not be limited to hardware verification. We do this by reviewing the application of PSL to various fields, such as missile interception algorithms, generated code for concurrent systems, or policies from policy-based middleware. There is a large amount of literature available about CTL and LTL use; this is also relevant for PSL since CTL and LTL are sub-languages of PSL. As PSL is an IEEE standard, we believe it can be used successfully for a wide variety of problems beyond hardware verification.

Acknowledgements The authors wish to thank Cindy Eisner for her helpful suggestions.

References

1. Dakshi Agrawal et al. Policy Management of Networked Systems and Applications. In *Proc. of 9th Intl. Symp. on Integrated Network Management*, IFIP/IEEE 2005.
2. S. Barner, Z. Glazberg, and I. Rabinovitz. Wolf—Bug Hunter for Concurrent Software Using Formal Methods. In *Proc. of 17th International Conference on Computer Aided Verification, LNCS 3576*, Springer, 2005.
3. S. Barner and I. Rabinovitz. Efficient symbolic model checking of software using partial disjunctive partitioning. *CHARME, LNCS 2860*, 2003.
4. I. Beer et al. RuleBase: An Industry-Oriented Formal Verification Tool. In *Proc. of the 33rd Design Automation Conference*, 1996.
5. G. Booch, J. E. Rumbaugh, and I. Jacobson. *Unified Modeling Language User Guide*. Addison-Wesley, 1999.
6. D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the Association for Computing Machinery*, 30(2), 1983.
7. Cindy Eisner, Dana Fisman. *A Practical Introduction to PSL.* Springer, August 2005.
8. Janees Elamkulam et al. Detecting Design Flaws in UML State Charts for Embedded Software, to *In Proc. of Haifa Verification Conference HVC 2006.* LNCS 4383, Springer, 2006.
9. E.M. Clarke and E.A. Emerson, Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic. In *Proc. of Workshop on Logics of Programs, LNCS 131*, Springer, 1981.
10. D. Ghose. True Proportional Navigation with Maneuvering Target, *IEEE Trans. on Aerospace and Electronic Systems*, 1994.
11. C.-F. Lin. Modern Navigation, Guidance and Control Processing. Prentice Hall, 1991.
12. M. Moulin, L. Gluhovsky, and E. Bendersky. Formal Verification of Maneuvering Target Tracking. *Proc. of the AIAA Conf. of Guidance, Navigation and Control*, Austin, TX, 2003.
13. A. Pnueli. A Temporal Logic of Concurrent Programs. In *Theoretical Computer Science*, Vol 13, 1981.
14. M. L. Steinberg. Comparison of Intelligent, Adaptive, and Nonlinear Flight Control Laws, *Journal of Guidance, Control and Dynamics*, 2001.
15. A. van der Schaft, H. Schumacher. An Introduction to Hybrid Dynamical Systems. Springer, 2000. V.251 of Lecture Notes in Control and Information Sciences.
16. A. Wasowski. Flattening Statecharts without Explosions. In *Proc. of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, 2004.
17. S. Wright, R. Chadha, and G. Lapiotis (eds): *Special Issue on Policy Based Networking*, *IEEE Networking 16*, 2002.
18. Emmanuel Zarpas. A Case Study: Formal Verification of Processor Critical Properties, *Correct Hardware Design and Verification Methods: CHARME 2005, LNCS 3725*, Springer 2005.
19. *IBM Tivoli System Automation for Multi-platforms, Guide and Reference, version 1.2,* IBM, 2004.

20. *IBM Tivoli System Automation for Multi-platforms, Base Component Reference, version 2.1.1*, 2006.
21. *IEEE Standard for Property Specification Language IEEE Std. 1850-2005*, 2005.
22. RuleBase PE homepage.
http://www.haifa.il.ibm.com/projects/verification/RB_Homepage/index.html, 2006