Validating Assertion Language Rewrite Rules and Semantics with Automated Theorem Provers

Katell Morin-Allory, Marc Boulé, Dominique Borrione, and Zeljko Zilic, *Senior Member, IEEE*

***Abstract*—Modern assertion languages such as property specification language (PSL) and SystemVerilog assertions include many language constructs. By far, the most economical way to process the full languages in automated tools is to rewrite the majority of operators to a small set of base cases, which are then processed in an efficient way. Since recent rewrite attempts in the literature have shown that the rules could be quite involved, sometimes counterintuitive, and that they can make a significant difference in the complexity of interpreting assertions, ensuring that the rewrite rules are correct is a major contribution toward ensuring that the tools are correct, and even that the semantics of the assertion languages are well founded. This paper outlines the methodology for computer-assisted proofs of several publicly known rewrite rules for PSL properties. We first present the ways to express the PSL syntax and semantics in the prototype verification system (PVS) theorem prover, and then prove or disprove the correctness of over 50 rewrite rules published without proofs in various sources in the literature. In doing so, we also demonstrate how to circumvent known issues with PSL semantics regarding the never and eventually! operators, and offer our proposals on assertion language semantics.**

***Index Terms*—Assertion languages, automated theorem provers, language semantics, proofs, rewrite rules.**

I. Introduction

THE USE of assertions in hardware [1] has increased rapidly in the past decade, as it has helped to deal with the increased challenges in hardware verification. Assertions offer a means of adding specifications incrementally to a design, and also offer a means of verifying a design with its expected behavior. Notwithstanding the traditional dynamic verification techniques (such as simulation), assertions are also used in formal verification approaches such as model checking. With the standardization of the two main assertion languages, namely property specification language (PSL) and SystemVerilog assertions (SVA), the industry now has a formally defined and industrially robust means of specifying properties, assumptions and design intent in hardware designs.

Manuscript received June 23, 2009; revised November 29, 2009 and March 12, 2010. Date of current version August 20, 2010. This paper was recommended by Associate Editor R. F. Damiano.

K. Morin-Allory is with the Grenoble Institute of Technology (Grenoble INP), Grenoble 38031, France (e-mail: katell.morin-allory@imag.fr).

D. Borrione is with Polytech Department, Joseph Fourier University, Grenoble 38031, France (e-mail: dominique.borrione@imag.fr).

M. Boulé is with the École de Technologie Supérieure, Montreal, QC H3C 1K3, Canada (e-mail: marc.boule@etsmtl.ca).

Z. Zilic is with the Department of Electrical Engineering, McGill University, Montreal, QC H3A 2A7, Canada (e-mail: zeljko.zilic@mcgill.ca).

Digital Object Identifier 10.1109/TCAD.2010.2049150

Devising circuit representations of assertions allows the assertion based verification paradigm to reach beyond the usual verification tools that commonly interpret such assertions. When converted to assertion *checkers* [2]–[4], also called monitors [5], the assertion verification capability can be used in hardware emulation and post-fabrication debugging [6], [7] to augment visibility and to assist in locating errors. Checkers can also allow simulators and formal verification tools to support assertions, and can be instrumental in devising the test sequences used in dynamic verification [8].

The HORUS checker generator has had its modular approach to property construction formally proved by automated theorem proving [3], [9]. In the modular approach to checker generation, each assertion operator is implemented as its own separate sub-module (or sub-circuit), and is connected to other modules to form a checker for a complete assertion. The sub-modules are interconnected according to the syntax tree of the assertion, in a recursive process. Once the complete circuit is constructed, a checker is produced and the output signals can be observed during verification for finding errors. The functional correctness of the method used is proved using formal methods in the PVS proof system. The library of components for property operators is proved correct, and then the interconnection scheme is proved correct, and by induction the checkers that are generated are proved correct.

Due to the richness of modern assertion languages, the use of rewrite rules is essential in developing compact and efficient checker generator tools. Rewrite rules have been used in the MBAC checker generator [10] to help implement the various forms of property operators in PSL. Although the core of the checker generator is based on automata for expressing properties, rewrite rules play a key role in compiling a large part of the language.

Cimatti *et al.* present the theoretical foundations of a verification tool [11] where the use of expression rewriting is instrumental to achieve automata suitable for formal verification. In their approach, the rewrite rules are devised to favor the part of the language that is better suited for the intended infinite trace automata representations.

Rewrite rules for PSLs simple subset have also been developed by Singh and Garg [12] for an early version of PSL. The goal of these rewrite rules is to convert various assertion forms into regular expression implications, for further use by simulation or formal verification engines. In that work, along with the work by Cimatti *et al.*, the rewrite rules are mainly developed for linear temporal logic (LTL) operators

and sequential regular expressions. In general, rewrite rules are often counterintuitive, and their correctness is not guaranteed, as we will demonstrate in this paper by disproving some of the published rewrite rules.

Previous work exploring the use of automated theorem provers to ascertain assertion language semantics is relatively limited. We found the work of M. Gordon particularly inspiring, in which he performed a "deep embedding" of PSL in the higher order logic (HOL) proof assistant, and used HOL to demonstrate theorems about the semantics. From this formalization he was able to build, within the logic, a generator to produce correct-by-construction mathematical observers for PSL properties [13]. The direct execution of the PSL semantics in HOL complies with the language definition by construction, but the resulting observers are too inefficient to be used in large designs. In contrast, in our research we formally prove the principles on which production-quality checkers can be generated for the wide variety of PSL operators (MBAC rewrite rules), along with a proved correct implementation for a kernel of PSL primitives (HORUS tool).

Other work by Claessen and Martensson proposed an operational semantic definition guided by the structure of weak PSL formulas [14], and showed some inconsistencies in the interpretation of a special class of regular expressions. Their goal was to investigate alternative semantic definitions to the trace semantics, as a way to correct problems with the early versions of PSL; if and how this was useful to guide an implementation of property checkers has not been published.

The PSL language reference contains a set of restrictions to help create properties that are more suitable for simulation and dynamic verification, akin to the simulation–friendly restrictions proposed for the generalized symbolic trajectory evaluation (GSTE) monitors in [15]. In the checkers developed for GSTE assertion graphs, certain constructs are avoided or restricted in scope, for hardware implementation reasons.

PSL's simple subset guidelines [16, Sec. 4.4.4] are intended to ensure that time advances monotonically along a single path, and thus help express assertions that are easier to grasp given the complex temporal logic that can be specified in such languages. The rewrite rules appearing in this paper follow the PSL simple subset (PSL-SS) guidelines, which are presented in the next section; however, the methods proposed apply to the vast majority of operators found in PSL and SVA.

This paper extends previous research devoted to checker generators, assertion language semantics, and their proofs of correctness [17]. While we deal mainly with proving the correctness of rewrite rules for PSL using the automated reasoning framework applied to the HORUS tool, we also provide insights into subtleties of correct checker synthesis in general, as well as the assertion language definition itself.

The main contribution of this paper is to show how an automated theorem prover can be used to ensure the correctness of assertion language equivalences and rewrite rules. We apply our methods to three sets of rewrite rules found in the literature (over 50 rules in total), and attempt to prove that they are well founded and semantically correct. Incorrect rules or minor semantic issues with rules can have an important consequence in electronic design automation tools that process assertions. The topics and results contained in this paper can be a helpful reference for tool designers or anyone with an interest in assertion languages.

The proofs we perform are founded on a higher-order semantic definition of PSL in the logic of PVS [18] and on the use of the PVS proof assistant to mechanize the formal reasoning. For the few cases where the proofs do not succeed, we provide counterexamples. We also highlight problems with the semantics of two operators in the PSL simple subset, and propose a formally justified revision for these guidelines.

This paper is organized as follows. Section II provides a concise description of PSLs syntax and semantics along with its simple subset guidelines, such that the paper is self contained. In Section III, we present the sets of rewrite rules to be proved. Section IV shows the general strategy used to perform the proofs in PVS, and in Section V, the proof results are detailed and analyzed. In Section VI, we outline our proposed improvements to the simple subset guidelines.

II. Assertion Languages and PSL

Ever since IBM donated their Sugar language to Accellera for forming the first industry standard for specifying hardware assertions, PSL has continued to develop and evolve into many verification applications. Standardization of the PSL language by the IEEE in 2005 has resulted in formally defined syntax and semantics, as documented in the PSL Language Reference Manual [16]. Other sources offer more pragmatic treatments of PSL [19], and can also help to grasp the intricacies of such languages.

The Verilog flavor of PSL is used throughout the paper; however, the proofs and techniques are not restricted to PSL since many concepts can easily apply to the proof of SVA rewrite rules, for example.

A. Syntax and Semantics of PSL

The logic of the IEEE 1850 PSL standard is defined with respect to a nonempty set P of atomic propositions, and a set B of Boolean expressions over P. In practice, the atomic propositions correspond to conditions and evaluations over the registers and wires of the design. The semantics of PSL is defined with respect to finite and infinite words over the alphabet $\Sigma = 2^P \cup \{\top, \bot\}$. In dynamic verification, a word v corresponds to a trace of execution of the design under test, and a letter l from v corresponds to a valuation of propositions in a given execution cycle. The i^{th} letter in a word is designated v^{i-1}, and the length of a word is noted as $|v|$. A sub-word of v is designated $v^{i,\cdots,j}$, where the sub-word is the range of letters from v^i to v^j. A suffix of a word can also be expressed as $v^{i\cdot\cdot}$, and is understood to mean the word starting at letter v^i. The notation l^ω is used to designate an infinite word composed of the letter l, and $\overline{v}$ is v with every $\top$ replaced by $\bot$ and vice versa.

The semantics of Boolean expressions is the base case, and is defined using the symbol $\models$. We say that a letter l satisfies a Boolean expression b, noted $l \models b$. The semantics of Boolean satisfaction is defined as follows: for every $l \in 2^P$, $p \in P$, and $b \in B$, we have (using Verilog notation):

1) $l \models p \quad \Leftrightarrow \quad p \in l$;
2) $l \models !b \quad \Leftrightarrow \quad l \not\models b$;
3) $l \models b_1 \,\&\, b_2 \quad \Leftrightarrow \quad l \models b_1$ and $l \models b_2$.

For example, if the set P of atomic propositions is {a, b, c}, consider a letter l consisting of the state ⟨a, c⟩ (atomic propositions not listed are assumed false). For the Boolean expressions b_1 = c and b_2 = b & c we have

$$l \models b_1 \text{ (i.e., } \langle \mathsf{a}, \mathsf{c} \rangle \text{ satisfies c)}$$
$$l \not\models b_2 \text{ (i.e., } \langle \mathsf{a}, \mathsf{c} \rangle \text{ does not satisfy b \& c)}$$

In this paper, the Booleans true and false represent the Verilog constants 1'b1 and 1'b0, respectively. These constants are such that for every letter l, we have $l \models$ true and $l \not\models$ false. Further, two special symbols $\top$ and $\bot$ are defined such that for every Boolean expression b, $\top \models b$, and $\bot \not\models b$; however, these symbols are only used in the semantics and are not modeled in the design itself.

Sequential extended regular expressions (SEREs) (sequences) specify temporal chains of events of Boolean expressions on finite words, while properties express temporal operators over SEREs on finite or infinite words. The property layer adds additional temporal operators over sequences and Booleans, to allow extensive specification capabilities. In PSL, SEREs are defined as follows [16].

Definition 1: If b is a Boolean expression and r is a SERE, the following expressions define the *syntax of SEREs*.

- b
- $\{r\}$
- $r_1 ; r_2$
- $r_1 : r_2$
- $r_1 \mid r_2$
- r_1 && r_2
- [*0]
- r[*]

The ";" and ":" operators are concatenation and fusion (overlapped concatenation), respectively. The "|" and "&&" operators represent SERE "or" and SERE length-matching "and," respectively. The [*0] symbol is the empty SERE, and more generally the [*] operator expresses the Kleene closure of SERE r (matching zero or more occurrences of r).

Definition 2: The *semantics of tight satisfaction of SEREs*, denoted $\models$, is defined as follows.

- $v \models b$ iff $|v| = 1$ and $v^0 \models b$
- $v \models \{r\}$ iff $v \models r$
- $v \models r_1 ; r_2$ iff $\exists v_1, v_2$ s.t. $v = v_1 v_2$, $v_1 \models r_1$ and $v_2 \models r_2$
- $v \models r_1 : r_2$ iff $\exists v_1, v_2, l$ s.t. $v = v_1 l v_2$, $v_1 l \models r_1$ and $l v_2 \models r_2$
- $v \models r_1 \mid r_2$ iff $v \models r_1$ or $v \models r_2$
- $v \models r_1$ && r_2 iff $v \models r_1$ and $v \models r_2$
- $v \models$ [*0] iff $v = \epsilon$
- $v \models r$[*] iff either $v \models$ [*0] or $\exists v_1, v_2$ s.t. $v_1 \neq \epsilon$, $v = v_1 v_2$, $v_1 \models r$ and $v_2 \models r$[*]

PSL defines additional syntactic "sugaring" operators, such as non-length matching intersection (&), and various forms of repetition ([*k], [*i:j], [=] and [–>]). Additional syntactic "sugaring" operators in PSL simplify the writing of assertions, but do not add expressive power to the language. The PSL SERE sugaring operators that appear throughout this paper are shown below.

Definition 3: If b is a Boolean expression; r is a SERE; i, j are nonnegative integers and k, l are positive integers with $j \geq i$ and $l \geq k$, then the following are *SERE sugaring operators*.

- r[+] $\stackrel{\text{def}}{=} r ; r$[*]
- r[*k] $\stackrel{\text{def}}{=} r ; r ; \ldots ; r$ (k times)
- r[*i:j] $\stackrel{\text{def}}{=} r$[*$i$] | ... | r[*j]
- b[–>] $\stackrel{\text{def}}{=} \{(!b)[*] ; b\}$
- b[–>k] $\stackrel{\text{def}}{=} \{b[->]\}[*k]$
- b[–>k:l] $\stackrel{\text{def}}{=} \{b[->]\}[*k{:}l]$
- r_1 & r_2 $\stackrel{\text{def}}{=}$ $\{\{r_1\}$ && $\{r_2$;true[*]$\}\}$ | $\{\{r_1$;true[*]$\}$ && $\{r_2\}\}$

The [*k] and [*i:j] operators are known as repetition count and repetition range. The first three operators can be used without the SERE r, in which case r = true is implied. The [–>] operator is known as goto repetition, and causes a matching of its Boolean argument at its first occurrence.

Properties in PSL allow other kinds of temporal operators to be specified over sequences and Booleans. Below are the foundation language property operators defined in PSL. The separating line is used only to help categorize the operators for use in Section III-A.

Definition 4: If b is a Boolean expression, r is a SERE and p is a property, i, j are nonnegative integers and k, l are positive integers with $j \geq i$ and $l \geq k$, then PSL *foundation language (FL) properties* are defined as follows.

- b
- (p)
- r!
- !p
- p abort b
- r
- p_1 <–> p_2
- p_1 && p_2
- r |–> p

- r |=> p
- always p
- b || p
- p_1 until p_2
- p_1 until! p_2
- p_1 before p_2
- p_1 before! p_2
- next p
- next[i](p)
- next_a[i:j](p)
- next_e[i:j](p)
- next_event(b)(p)
- next_event(b)[k](p)
- next_event_a(b)[k:l](p)
- next_event_e(b)[k:l](p)
- eventually! p
- never p
- b –> p
- p_1 until_ p_2
- p_1 until!_ p_2
- p_1 before_ p_2
- p_1 before!_ p_2
- next! p
- next![i](p)
- next_a![i:j](p)
- next_e![i:j](p)
- next_event!(b)(p)
- next_event!(b)[k](p)
- next_event_a!(b)[k:l](p)
- next_event_e!(b)[k:l](p)

The semantics of (p) is the same as the semantics of p since the parentheses are used only for grouping. The LTL operators X, X!, G, F, U and W in PSL are equivalent to the operators next, next!, always, eventually!, until! and until, respectively; for simplicity, they are omitted in Definition 4. The semantics of the remaining property operators is given next, as defined in Appendix B in the PSL Specification [16]. The items below based on the *iff* symbol are defined as the basic operators in PSL, whereas items based on the $\stackrel{\text{def}}{=}$ symbol are defined as syntactic sugaring.

Definition 5: The *semantics of PSL properties* is defined using the satisfaction criterion $v \models p$, which means that property p holds in trace v (trace v satisfies property p).

- $v \models b$ iff $|v| = 0$ or $v^0 \models b$
- $v \models !p$ iff $\overline{v} \not\models p$
- p_1 <–> p_2 $\stackrel{\text{def}}{=}$ (p_1 –> p_2) && (p_2 –> p_1)
- $v \models p$ abort b iff either $v \models p$ or $\exists j < |v|$ s.t. $v^j \models b$ and $v^{0..j-1} \top^\omega \models p$

- $v \models p_1$&& p_2 iff $v \models p_1$ and $v \models p_2$
- $v \models r!$ iff $\exists j < |v|$ s.t. $v^{0..j} \models\!\models r$
- $v \models r$ iff $\forall j < |v|$, $v^{0..j}\top^{\omega} \models r!$
- $v \models r$|–> p iff $\forall j < |v|$ s.t. $\overline{v}^{0..j} \models\!\models r$, $v^{j..} \models p$
- r|=> $p \stackrel{\text{def}}{=}$ {r; true} |–> p
- eventually! $p \stackrel{\text{def}}{=}$ [true U p]
- always $p \stackrel{\text{def}}{=}$!eventually! (!p)
- never $p \stackrel{\text{def}}{=}$ always !p
- p_1 || $p_2 \stackrel{\text{def}}{=}$!(!p_1 && !p_2)
- p_1 –> $p_2 \stackrel{\text{def}}{=}$!p_1 || p_2
- p_1 until $p_2 \stackrel{\text{def}}{=}$ (p_1 until! p_2) || always p_1
- $v \models p_1$ until! p_2 iff $\exists k < |v|$, $v^{k..} \models p_2$ and $\forall j < k$, $v^{j..} \models p_1$
- p_1 until_ $p_2 \stackrel{\text{def}}{=}$ p_1 until (p_1 && p_2)
- p_1 until!_ $p_2 \stackrel{\text{def}}{=}$ p_1 until! (p_1 && p_2)
- p_1 before $p_2 \stackrel{\text{def}}{=}$!p_2 until (p_1 && !p_2)
- p_1 before_ $p_2 \stackrel{\text{def}}{=}$!p_2 until p_1
- p_1 before! $p_2 \stackrel{\text{def}}{=}$!p_2 until! (p_1 && !p_2)
- p_1 before!_ $p_2 \stackrel{\text{def}}{=}$!p_2 until! p_1
- $v \models$ next! p iff $|v| > 1$ and $v^{1..} \models p$
- next $p \stackrel{\text{def}}{=}$!next! (!p)
- next[i](p) $\stackrel{\text{def}}{=}$ next next ... next p (i times)
- next![i](p) $\stackrel{\text{def}}{=}$ next! next! ... next! p (i times)
- next_a[i:j](p) $\stackrel{\text{def}}{=}$ next[i](p) && ... && next[j](p)
- next!_a[i:j](p) $\stackrel{\text{def}}{=}$ next![i](p) && ... && next![j](p)
- next_e[i:j](p) $\stackrel{\text{def}}{=}$ next[i](p) || ... || next[j](p)
- next!_e[i:j](p) $\stackrel{\text{def}}{=}$ next![i](p) || ... || next![j](p)
- next_event(b)(p) $\stackrel{\text{def}}{=}$!b until (b && p)
- next_event!(b)(p) $\stackrel{\text{def}}{=}$!b until! (b && p)
- next_event(b)[k](p) $\stackrel{\text{def}}{=}$ next_event(b)(next next_event(b) ... (next next_event(b)(p))...) (k times)
- next_event!(b)[k](p) $\stackrel{\text{def}}{=}$ next_event!(b)(next! next_event!(b) ... (next! next_event!(b)(p))...) (k times)
- next_event_a(b)[k:l](p) $\stackrel{\text{def}}{=}$ next_event(b)[k](p) && ... && next_event(b)[l](p)
- next_event_a!(b)[k:l](p) $\stackrel{\text{def}}{=}$ next_event!(b)[k](p) && ... && next_event!(b)[l](p)
- next_event_e(b)[k:l](p) $\stackrel{\text{def}}{=}$ next_event(b)[k](p) || ... || next_event(b)[l](p)
- next_event_e!(b)[k:l](p) $\stackrel{\text{def}}{=}$ next_event!(b)[k](p) || ... || next_event!(b)[l](p)

The verification layer in PSL defines several directives to indicate what a property is intended for. In particular, assert is used to specify that a given property must hold. No semantics is defined for this operator, as its interpretation depends on the type of verification tool being used. For example, in static property checking, a formal verification engine could report that the property holds or does not hold. In dynamic verification, a simulator could flag instances where the property did or did not hold throughout the simulation trace.

B. PSLs Simple Subset Guidelines

Below are the main property operators defined in PSL, with the simple subset guidelines applied. A total of eleven such guidelines are introduced [16], which help to create properties that are more amenable to simulation, or any form of dynamic verification for that matter. According to the PSL Issues List (item #99, simple subset issues [20]), the overlapping until operator can allow a full property on the left-side, as opposed to only a Boolean expression as originally indicated in the simple subset guidelines. This flexibility has been taken into account in our work. The separating line is also used to help categorize the operators for use in Section III-A.

Definition 6: If b is a Boolean expression, r is a SERE and p is a property, i, j are nonnegative integers and k, l are positive integers with $j \geq i$ and $l \geq k$, then PSL FL properties are defined as follows in *PSLs simple subset*. Operators not listed remain as defined in Definition 4.

• !b	• b_1 <–> b_2
• eventually! r	• never r
• b || p	• b –> p
• p until b	• p until_ b
• p until! b	• p until!_ b
• b_1 before b_2	• b_1 before_ b_2
• b_1 before! b_2	• b_1 before!_ b_2
• next_e[i:j](b)	• next_e![i:j](b)
• next_event_e(b_1)[k:l](b_2)	• next_event_e!(b_1)[k:l](b_2)

To summarize the modifications, some or all of the property arguments appearing in the operators above were reduced to either sequences or Booleans.

III. Rewrite Rules

In languages such as PSL, a few base operators define the core of the language while supplemental operators are added as a syntactical convenience for writing expressions. For example, in *pure* LTL, the expression X[4] p, which says that p must hold in 4 cycles (states), would actually have to be written as: X X X X p.

Rewrite rules often take the form $x \rightarrow y$, to express that when expression x is encountered, it is syntactically replaced by expression y. The kernel of the tool does not even need to support the form in the left side, since it gets rewritten to more basic operators in the right side. In this paper, the $\stackrel{\rightarrow}{=}$ symbol is used to show that both side of the rule are to be considered equivalent (upon confirmation in the proofs), but with a preferred direction to be used as a rewrite rule. When the right-hand sides of the rewrite rules are not terminal, they are typically rewritten using other rules, until no more rewrites apply and the base cases of sequences, Boolean expressions and properties are reached. When doing so, proper care must be taken to ensure that the set of rules is *terminating*, and that no infinite cycle of application of the rules can exist.

In this section, we present the three sets of rewrite rules selected for analysis in our PSL embedding in PVS. The goal will be to prove or disprove each rules validity, and more generally to investigate the soundness of the language constructs.

A. MBAC Rewrite Rules

As presented previously, PSL properties are based on LTL, with the addition of a wide array of sugaring operators. Most of these sugaring operators are defined as language equivalences in the PSL specification [16]. Because of the restrictions imposed by the simple subset, these equivalences

can generally not be used as rewrite rules. For example, consider the definition of the never operator with a SERE as its argument. If we apply the definition

$$\text{never } r \stackrel{\text{def}}{=} \text{always } !r$$

a verification tool made for dynamic verification based on the simple subset does not support the negation of a SERE, since by definition, negating a SERE is not defined in the simple subset (Definition 6). In other words, we need a definition (or rewrite rule) for the never operator that remains in the realm of the simple subset.

For this reason, a set of rewrite rules particularly suited to dynamic verification was developed and used in the MBAC checker generator [10]. This allows the majority of property operators to be rewritten to a smaller set of base cases, for which specialized automata-based algorithms were developed.

The nine operators shown above the separating line in Definition 4 are the base cases that have specific automata implementations in [10], thus no rewrite rules are used. Two of the base cases are reduced in the simple subset, as shown above the separating line in Definition 6. As such, negation and equivalence (<–>) apply only to Booleans and are therefore implicitly handled in the semantics of Booleans.

The remaining thirty property operators in Definition 4 (below the separating line), of which sixteen are modified in the simple subset (below the line in Definition 6), are implemented using rewrite rules in the MBAC checker generator [4], [10]. The rules are presented below and are grouped into three themes, according to a common characteristic they share.

Proposition 1: MBAC Rewrite Rules Based on Implication.

R1: $b \mid\mid p \quad \vec{=} \quad \{!b\} \mid\!\!\rightarrow p$
R2: $b \rightarrow p \quad \vec{=} \quad \{b\} \mid\!\!\rightarrow p$
R3: $r \mid\!\!=\!\!> p \quad \vec{=} \quad \{r \,;\, \text{true}\} \mid\!\!\rightarrow p$
R4: $\text{always } p \quad \vec{=} \quad \{[+]\} \mid\!\!\rightarrow p$
R5: $\text{never } r \quad \vec{=} \quad \{[+] : r\} \mid\!\!\rightarrow \text{false}$
R6: $p \text{ until } b \quad \vec{=} \quad \{(!b)[+]\} \mid\!\!\rightarrow p$
R7: $p \text{ until\_ } b \quad \vec{=} \quad \{\{(!b)[+]\} \mid \{b[\rightarrow]\}\} \mid\!\!\rightarrow p$
R8: $\text{next\_event\_a}(b)[k{:}l](p) \quad \vec{=} \quad \{b[\rightarrow k{:}l]\} \mid\!\!\rightarrow p$

Rule R1 is based on the fact that if the Boolean expression is false, then the argument property p must be true; otherwise the entire property is automatically true. The implication in R2 can be rewritten using a suffix implication because a Boolean expression can be easily expressed as a sequence. The R3 rewrite rule for non-overlapped property implication follows from its definition in Appendix B in [16]. When a property must always be true (R4), it can be seen as the consequent of a suffix implication with a perpetually active start condition. When a sequence must not occur (R5), a property that fails instantly is triggered upon detection of the sequence.

The until operator states that property p must be true on each cycle, up-to, but not including, b being true. In R6, the implication has the effect of sending an activation to start checking p for each cycle of consecutive $!b$'s. In the run-time interpretation semantics for the until operator in [10], the property is allowed to fail multiple times for a given activation when b is continuously false. Implementing the overlapped form of until (R7) is done by adding another condition for the property p, namely that it must also hold for the cycle in which the Boolean expression b is true.

The next_event_a property in R8 states that all occurrences of the next event within the specified range must see the property be true. This can be modeled using a goto repetition with a range, as an antecedent to the property via suffix implication. This sends an activation to check the property each time b occurs within the specified range after the current property received its activation.

Proposition 2: MBAC Rewrite Rules Based on Sequences.

R9: $\text{eventually! } r \quad \vec{=} \quad \{[+] : r\}!$
R10: $b_1 \text{ before } b_2 \quad \vec{=} \quad \{(!b_1 \& !b_2)[*] \,;\, (b_1 \& !b_2)\}$
R11: $b_1 \text{ before! } b_2 \quad \vec{=} \quad \{(!b_1 \& !b_2)[*] \,;\, (b_1 \& !b_2)\}!$
R12: $b_1 \text{ before\_ } b_2 \quad \vec{=} \quad \{(!b_1 \& !b_2)[*] \,;\, b_1\}$
R13: $b_1 \text{ before!\_ } b_2 \quad \vec{=} \quad \{(!b_1 \& !b_2)[*] \,;\, b_1\}!$
R14: $\text{next\_event\_e}(b_1)[k{:}l](b_2) \quad \vec{=} \quad \{b_1[\rightarrow k{:}l] : b_2\}$
R15: $\text{next\_event\_e!}(b_1)[k{:}l](b_2) \quad \vec{=} \quad \{b_1[\rightarrow k{:}l] : b_2\}!$

In the set of rules above, the common theme is to express the behavior of the operator on the left side using only a sequence. The sequence replaces the property, and thus also appears at the property level. The first rewrite (R9) expresses the eventuality as a strong sequence (!) which can begin at any cycle, hence the fusion with [+]).

The before family of properties in R10 to R13 can also be modeled by sequences. The overlapped versions state that b_1 must be asserted before or simultaneously with b_2. The next_event_e properties (R14 and R15) state that b_2 should be asserted at least once in the specified range of next events of b_1. This behavior is modeled by a goto repetition that is fused with the consequent. Once the b_2 consequent is observed in the proper range, the sequence has completed and will not indicate a failure. The strong versions of these properties are created by using strong sequences.

Proposition 3: MBAC Rewrites Based on Property Variations.

R16: $p \text{ until! } b \quad \vec{=} \quad (p \text{ until } b) \;\&\&\; (\{b[\rightarrow]\}!)$
R17: $p \text{ until!\_ } b \quad \vec{=} \quad (p \text{ until\_ } b) \;\&\&\; (\{b[\rightarrow]\}!)$
R18: $\text{next } p \quad \vec{=} \quad \text{next}[1](p)$
R19: $\text{next! } p \quad \vec{=} \quad \text{next}![1](p)$
R20: $\text{next\_event}(b)(p) \quad \vec{=} \quad \text{next\_event}(b)[1](p)$
R21: $\text{next\_event!}(b)(p) \quad \vec{=} \quad \text{next\_event!}(b)[1](p)$
R22: $\text{next}[i](p) \quad \vec{=} \quad \text{next\_event}(\text{true})[i{+}1](p)$
R23: $\text{next}![i](p) \quad \vec{=} \quad \text{next\_event!}(\text{true})[i{+}1](p)$
R24: $\text{next\_a}[i{:}j](p) \quad \vec{=} \quad \text{next\_event\_a}(\text{true})[i{+}1 : j{+}1](p)$
R25: $\text{next\_a}![i{:}j](p) \quad \vec{=} \quad \text{next\_event\_a!}(\text{true})[i{+}1 : j{+}1](p)$
R26: $\text{next\_e}[i{:}j](b) \quad \vec{=} \quad \text{next\_event\_e}(\text{true})[i{+}1 : j{+}1](b)$
R27: $\text{next\_e}![i{:}j](b) \quad \vec{=} \quad \text{next\_event\_e!}(\text{true})[i{+}1 : j{+}1](b)$
R28: $\text{next\_event}(b)[k](p) \quad \vec{=} \quad \text{next\_event\_a}(b)[k{:}k](p)$
R29: $\text{next\_event!}(b)[k](p) \quad \vec{=} \quad \text{next\_event\_a!}(b)[k{:}k](p)$
R30: $\text{next\_event\_a!}(b)[k{:}l](p) \quad \vec{=} \quad \{b[\rightarrow l]\}! \;\&\&\; \text{next\_event\_a}(b)[k{:}l](p)$

The group of rules in Proposition 3 is based on rewriting to other various forms of property operators. The strong versions of the until properties (R16 and R17) are created by using the weak versions, and by adding a temporal obligation for the releasing condition to occur, namely b. This can be modeled by the strong single-goto ([–>]) of the Boolean condition b.

The R18 to R21 rewrites use a slightly more explicit form of next operators. These rules are based on the fact that when no count is specified, a count of 1 is implicit.

The family of rules in R22 to R27 is based on the fact that next_event is a more general case of next. The "+1" adjustment is required to handle the mapping of the Boolean true. When converting a next property to a next_event property, there is a slight nuance as to what constitutes the next occurrence of a condition. The next occurrence of a Boolean expression can be in the current cycle, whereas the plain next implicitly refers to the next cycle.

The strategy behind the R28 and R29 rewrites is to utilize the next_event_a form, with identical upper and lower bounds for the range. Rule R30 handles the strong version of the full next_event_a property. Similarly to the strong non-overlapped until property, it is rewritten using the weak version, to which a necessary completion criterion is conjoined. The addition of the strong goto sequence with the l bound indicates that for each start condition of the next_event_a, all l occurrences of the b event must occur.

B. Cimatti et al. Rewrite Rules

Rewrite rules play an important part in the compilation of PSL into symbolically represented nondeterministic Büchi automata [11], for subsequent use in assertion-based formal verification. In this approach, rewrite rules are used to reduce SERE conjunction operators to simpler forms, and also to help convert suffix implication operators into LTL operators. Of the rewrite rules used in the symbolic compilation of PSL [11], we selected a few non-obvious rules to prove their validity using the PVS-based proof strategy outlined in this paper.

The selected subset of rewrite rules appears in Proposition 4. We used our own numbering, as opposed to that in the original publication. The rewrites are presented in the Verilog "flavor" of PSL, and we use the $\vec{=}$ symbol to specify the direction of the rewrite rule. As before, b is a Boolean expression, r is a SERE and p is a PSL property.

Proposition 4: Cimatti *et al.* Rewrite Rules [11] (subset).

CR1: $b \,\&\, r[*] \;\vec{=}\; b \mid \{b : r[*]\}$
CR2: $b[*] \,\&\, r \;\vec{=}\; \{r\} \mid \{\{b[*] \,\&\&\, r\} ; b[*]\}$
CR3: $r_1[*] \,\&\, r_2 \;\vec{=}\; r_2 \mid r_1[*] \,\&\&\, \{r_2 ; \mathsf{true}[*]\}$
CR4: $r_1[*] \,\&\, r_2[*] \;\vec{=}\; r_1[*] \mid r_2[*]$
CR5: $\{r_1 : r_2\} \mid\!\!\rightarrow p \;\vec{=}\; \{r_1\} \mid\!\!\rightarrow (\{r_2\} \mid\!\!\rightarrow p)$
CR6: $\{r_1 ; r_2\} \mid\!\!\rightarrow p \;\vec{=}^{\dagger}\; \{r_1\} \mid\!\!\rightarrow \mathsf{next}\,(\{r_2\} \mid\!\!\rightarrow p)$
CR7: $\{r_1 \mid r_2\} \mid\!\!\rightarrow p \;\vec{=}\; (\{r_1\} \mid\!\!\rightarrow p) \,\&\&\, (\{r_2\} \mid\!\!\rightarrow p)$
CR8: $\{r ; b[*]\} \mid\!\!\rightarrow p \;\vec{=}^{\ddagger}\; \{r\} \mid\!\!\rightarrow ((\mathsf{next}\ !b)\ \mathsf{R}\ p)$
CR9: $\{b[*] ; r\} \mid\!\!\rightarrow p \;\vec{=}^{\ddagger}\; !b\ \mathsf{R}\ (\{r\} \mid\!\!\rightarrow p)$
CR10: $b \,\&\&\, \{r_1 \,\&\&\, r_2\} \;\vec{=}\; \{b \,\&\&\, r_1\} \,\&\&\, r_2$
CR11: $b \,\&\&\, \{r_1 : r_2\} \;\vec{=}\; \{b \,\&\&\, r_1\} \,\&\&\, r_2$
CR12: $b \,\&\&\, r[*] \;\vec{=}\; b \,\&\&\, r$
CR13: $b[*] \,\&\&\, r[*] \;\vec{=}\; \{b[*] \,\&\&\, r\}[*]$
CR14: $\{b_1[*] ; r_1\} \,\&\&\, \{b_2 ; r_2\} \;\vec{=}\; \{r_1 \,\&\&\, \{b_2 ; r_2\} \mid \{\{b_1 \,\&\&\, b_2\} ; \{\{b_1[*] ; r_1\} \,\&\&\, r_2\}\}$
CR15: $\{b_1[*] ; r_1\} \,\&\&\, \{b_2[*] ; r_2\} \;\vec{=}\; \{b_1 \,\&\&\, b_2\}[*];\{\{r_1 \,\&\&\, \{b_2[*];r_2\}\} \mid \{\{b_1[*] ; r_1\} \,\&\&\, r_2\}\}$

Here, † : requires $\epsilon \notin \mathcal{L}(r_1)$ and $\epsilon \notin \mathcal{L}(r_2)$ and ‡: requires $\epsilon \notin \mathcal{L}(r)$.

In the rules CR8 and CR9, the LTL operator R is used, but is not defined in PSL. The operator is known as the "Release" operator, and its semantics are as follows:

$$p_1\ \mathsf{R}\ p_2 \quad = \quad !(!p_1\ \mathsf{U}\ !p_2)$$

where U is the strong operator "until!" in PSL. The semantics of the Release operator was added to our semantics modeling of PSL in PVS, and was used accordingly in the proofs of rewrites CR8 and CR9.

C. Singh and Garg Rewrite Rules

Rewrite rules are used by Singh and Garg to transform expressions using the simple subset of PSL into SERE implication formulas. We retain eight rules which are presented as rules 3 to 10 in Proposition 5 below; the numbering corresponds to that of the original publication [12]. The rewrite rules are formatted to be consistent with the style employed thus far in this paper. The first two cases shown as rules DR1 and DR2 are not labeled as rewrite rules in the original publication, but are instead presented elsewhere in the text to explain language semantics. We believe these two cases are more interesting to analyze, since the original rules labeled 1 and 2 are trivial to prove.

Proposition 5: Singh and Garg Rewrite Rules [12] (subset).

DR1: $\mathsf{eventually!}\ r \;\vec{=}\; \{\mathsf{true}\} \mid\!\!\rightarrow \{[*];r\}!$
DR2: $\mathsf{never}\ r \;\vec{=}\; r \mid\!\!\rightarrow \{\mathsf{false}\}$
DR3: $b_1\ \mathsf{until!}\ b_2 \;\vec{=}\; \{\mathsf{true}\} \mid\!\!\rightarrow \{b_1[*];b_2\}!$
DR4: $b_1\ \mathsf{until}\ b_2 \;\vec{=}\; \{\mathsf{true}\} \mid\!\!\rightarrow \{b_1[*];b_2\}$
DR5: $\mathsf{next}(r_1 \mid\!\!\rightarrow r_2) \;\vec{=}\; \{\mathsf{true};r_1\} \mid\!\!\rightarrow \{r_2\}$
DR6: $(r_1 \mid\!\!\rightarrow r_2)\ \mathsf{until}\ b \;\vec{=}\; \{\{!b[+]\}:r_1\} \mid\!\!\rightarrow \{r_2\}$
DR7: $(r_1 \mid\!\!\rightarrow r_2)\ \mathsf{until!}\ b \;\vec{=}\; \{\mathsf{true}\} \mid\!\!\rightarrow \{\mathsf{true}[*];b\}!\ \&\&\ \{\{!b[+]:r_1\} \mid\!\!\rightarrow \{r_2\}$
DR8: $\mathsf{next}(r_1 \mid\!\!\rightarrow r_2!) \;\vec{=}\; \{\mathsf{true};r_1\} \mid\!\!\rightarrow \{r_2\}!$
DR9: $(r_1 \mid\!\!\rightarrow r_2!)\ \mathsf{until}\ b \;\vec{=}\; \{\{!b[+]:r_1\} \mid\!\!\rightarrow \{r_2\}!$
DR10: $(r_1 \mid\!\!\rightarrow r_2!)\ \mathsf{until!}\ b \;\vec{=}\; \{\mathsf{true}\} \mid\!\!\rightarrow \{\mathsf{true}[*];b\}!\ \&\&\ \{\{!b[+]:r_1\} \mid\!\!\rightarrow \{r_2\}!$

Although some rules presented in this section may appear intuitive, they are included for completeness and to allow us to compare relatively simple proofs with more difficult cases. For the rules that are not so intuitive, the computer assisted proofs help to confirm that they are well founded. With a successful proof, any such rule can be used with even greater confidence in any tool that supports PSL.

IV. Modeling and Proving the Correctness of the Rewrite Rules in PVS

To prove the correctness of the rewrite rules, we use the PVS theorem prover. PVS provides an integrated environment for the development and analysis of formal verification. It consists of a specification language, a number of predefined theories and a theorem prover. It is based on a typed higher order logic. The choice of PVS was motivated by the fact that the PSL semantics is expressed in second-order logic and thus directly represented by the PVS input formalism. In addition, many proof strategies are automated.

We first modeled the syntax of PSL in PVS: we defined a datatype for the Boolean expressions, sequential expressions

```
Bool_PSL:DATATYPE
Begin
  true_S:true_S?
  false_S:false_S?
  sig(a:string):sig?
  and_bool(b1:Bool_PSL,b2:Bool_PSL):and_bool?
  or_bool(b1:Bool_PSL,b2:Bool_PSL):or_bool?
  not_bool(b:Bool_PSL):not_bool?
END Bool_PSL

SERE:DATATYPE
Begin
  importing Bool_PSL_adt
  Bool(b:Bool_PSL):Bool?
  empty_S: empty_S?
  seq(left:SERE, right:SERE):seq?
  fusion(left:SERE, right:SERE):fusion?
  star(s1:SERE):star?
  or_sere(s1:SERE,s2:SERE):or_sere?
  and_double(s1:SERE,s2:SERE):and_double?
  ...
END SERE

FL:DATATYPE
Begin
  importing SERE_adt, Bool_PSL_adt
  Sere(s:SERE):Sere?
  Sere_strong(s:SERE):Sere_Strong?
  impl_ovlap(s1:SERE,f:FL):impl_rec?
  impl_notovlap(s1:SERE,f:FL):impl_notrec?
  always(f:FL):always?
  ...
  not_FL(f:FL):not_FL?
END FL
```

Fig. 1. Datatype definition in PVS.

and FL properties. We then modeled the semantics of each datatype by a polymorphic function Sem. This modeling mimics the semantics definition from the PSL reference-language manual, and could be automatically defined. The last step is to define the equivalence of the rewrite rules and to perform the proofs.

A. Syntax Modeling

Three main datatypes called $Bool\_PSL$, $SERE$ and FL, are defined to represent the Boolean expressions, sequential expressions and foundation language properties of PSL, respectively. SEREs are defined using Booleans, whereas properties are defined using Booleans and SEREs.

The datatype definitions (Fig. 1) are relatively intuitive to understand, and we will explain the formalism of the $Bool\_PSL$ definition in more detail. In PSL, a Boolean expression is built over a set of atomic propositions. $Bool\_PSL$ is declared as a type with six constructors: `true_S`, `false_S`,[1] `sig`, `and_bool`, `or_bool`, `not_bool`. The constructors `true_S` and `false_S` take no argument, they represent the constants true and false, respectively. The predicate recognizer `true_S?` (`false_S?`) holds for exactly those expressions of $Bool\_PSL$ that are identical to `true_S` (`false_S`). The constructor `sig` takes one argument of type string representing an atomic proposition name. The predicate recognizer `sig?` holds for exactly those expressions of $Bool\_PSL$ that are constructed using `sig`. These three constructors are the base cases of $Bool\_PSL$, while the last three constructors are used to define it recursively. The constructor `and_bool` takes two arguments of type $Bool\_PSL$. It represents the conjunction of Boolean expressions. Predicate `and_bool?` holds for expressions that are built using `and_bool`. The last two constructors are defined similarly to `and_bool`, they represent the disjunction and the negation of Booleans.

[1]We cannot directly use the Boolean type defined in PVS to represent `true_S`, and `false_S` in order to distinguish the syntax and the semantics. `true_S` and `false_S` are a syntactical definition, their semantics is given by a function, the image of which is the PVS Boolean type.

```
Trace_PSL: DATATYPE
Begin
  val(l:list[string]):val?
  top: top?
  bot: bot?
END Trace_PSL
```

Fig. 2. Datatype definition of the set of traces in PVS.

B. Semantics Modeling

Time is discrete in PSL, and we model it by the set of natural numbers $\mathbb{N}$. The datatype `Trace_PSL` (Fig. 2) represents the set $\{0, 1\}^P \cup \{\top, \bot\}$: an element of `Trace_PSL` is either $\top$, or $\bot$, or a list of strings (signal names), for the signals whose value is true.

A trace is a mapping from $\mathbb{N}$ to `Trace_PSL`. We denote by $\mathcal{T}$ the set of traces and $\mathcal{T}_{|\text{val}}$ the restriction of $\mathcal{T}$ to the traces constructed with the `val` constructor; $\mathcal{T}_{|\text{val}}$ represents fully defined traces containing no $\top$ nor $\bot$.

The semantics is defined recursively for each datatype. The semantics of Boolean expressions is modeled by a mapping

$$\begin{array}{rccc} \text{Sem}: & Bool\_PSL \times \mathbb{N} \times \mathcal{T} & \longrightarrow & \mathbb{B} \\ & b, t, v & \mapsto & v^t \models b \end{array}$$

The mapping evaluates $v^t \models b$, and its definition in PVS is given in Fig. 3. If the value of v^t is $\top$ ($\bot$), then Sem(b, t, v) is true (false). Otherwise it depends on the semantics of b. If b is an atomic proposition of the type `sig(a)`, its semantics is given by $\mathsf{a} \in v^t$. In PVS, we use a predefined predicate `member` applied on an element e and a list l that is true when e is in l. Fig. 3 illustrates the implementation in PVS. Since this mapping is recursively defined, it is necessary to show that the recursion stops. We defined a *measure* given by the size of the Boolean expression (the depth of its syntax tree). The type verification of PVS is used to show that the measure is well founded.

In the sequel, we concisely summarize the semantics modeling for sequences and properties and present their PVS representations, together with a few concrete examples.

The semantics of SEREs is modeled by a mapping

$$\begin{array}{rccc} \text{Sem}: & \mathcal{S} \times \mathbb{N} \times \mathbb{N} \cup \{-1\} \times \mathcal{T} & \longrightarrow & \mathbb{B} \\ & s, t_0, T, v & \mapsto & v^{t_0 \ldots T} \models s \end{array}$$

The parameter t_0 represents the starting cycle of the trace, and T represents the end of the trace. An empty trace is specified when T is equal to $t_0 - 1$. The translation from the standard to the modeling is straightforward and we illustrate it using an example.

```
sem(b:Bool_PSL,t:nat, v:[nat->Trace_PSL]):
                          RECURSIVE Boolean=
CASES v(t) of
   top: true,
   bot: false,
   val(l):
      CASES b of
        true_S: true,
        false_S: false,
        sig(a):  member(a,l),
        and_bool(b1,b2): sem(b1,t, v)
                     and sem(b2,t,v),
        or_bool(b1,b2): sem(b1,t,v)
                     or sem(b2,t,v),
        not_bool(b1): not  sem(b1,t,v)
      ENDCASES
ENDCASES
MEASURE ( size(b));
```

Fig. 3. Definition of the Boolean semantics in PVS.

Example 1: The PVS semantics of $s[*]$, known as the SERE Kleene star operator, mirrors the semantics of the SERE Kleene star operator presented in Definition 2, and is captured in PVS as follows:

$$\begin{aligned} &\mathrm{Sem}(s[*], t_0, T, v) = \\ &\quad \mathrm{Sem}([*0], t_0, T, v) \\ \vee \quad &\exists t_1 \in \mathbb{N} : t_0 \le t_1 \le T \wedge \mathrm{Sem}(s, t_0, t_1, v) \\ &\quad \wedge \mathrm{Sem}(s[*], t_1 + 1, T, v) \end{aligned}$$

Moving on to the property layer in PSL, the semantics of FL operators is modeled by a mapping Sem

$$\begin{array}{ccc} \mathrm{Sem} : FL \times \mathbb{N} \times \mathbb{N} \cup \{-1\} \times \mathcal{T} \times \mathbb{B} & \longrightarrow & \mathbb{B} \\ f, t_0, T, v, inf & \mapsto & v^{t_0 \cdots} \models f \text{ if } inf \\ & & v^{t_0 \cdots T} \models f \text{ else} \end{array}$$

The parameter *inf* indicates whether the semantics is defined on a finite or infinite trace v. The definition of the semantics of FL properties is illustrated with an example.

Example 2: The semantics of f_1 until! f_2 is taken directly from the standard and is expressed in PVS as follows:

$$\begin{aligned} &\mathrm{Sem}(f_1 \text{ until! } f_2, t_0, T, v, inf) = \exists k \in \mathbb{N} : \\ &\quad t_0 \le k \wedge (\neg inf \Rightarrow k \le T) \\ \wedge \quad &\mathrm{Sem}(f_2, k, T, v, inf) \\ \wedge \quad &\forall j, t_0 \le j < k \Rightarrow \mathrm{Sem}(f_1, j, T, v, inf) \end{aligned}$$

Following this method, the PSL syntax and semantics was specified in approximately 400 lines of PVS code. Next we show how to model the rewrite rules in PVS. The rewrites are entered as theorems and constitute the objective of the interactive proof sessions.

C. Theorem Modeling

A theorem defines the equivalence of each rewrite rule in PVS. The equivalence is proved regardless of the length of the trace, and for traces that are fully defined (with no $\top$ nor $\bot$). The letter $\top$ or $\bot$ may be introduced in the process of proving the semantic equivalence of a rewrite rule; however, both the circuit under test and our checkers do not rely on these symbols. For static verification, the rules remain to be validated when $\top$ and $\bot$ are to appear in traces : for example, the $\top$ symbol appears in the semantics of the abort operator. In dynamic verification, the abort does not extend traces with $\top^\omega$ but instead discharges the evaluation of a subproperty using other mechanisms. In our checkers, the abort is implemented in hardware [10] and is compatible with the rewrite rules.

The modeling for rule R16 (strong until) is shown as an example below.

Example 3: The rewrite rule for the strong until! operator is based on the weak until operator, with a conjunction to the strong goto-repetition matching of the eventuality condition, namely $\{b[->]\}!$. The theorem is modeled in PVS as follows, using our semantics definition shown previously

$$\begin{aligned} &\forall f \in FL, b \in Bool\_PSL, t_0 \in \mathbb{N}, T > t_0 - 1, v \in \mathcal{T}_{|\mathrm{val}} \\ &\quad \mathrm{Sem}(f \text{ until! } b, t_0, T, v, inf) \\ \Leftrightarrow \quad &\mathrm{Sem}((f \text{ until } b) \,\&\&\, (\{b[->]\}!), t_0, T, v, inf) \end{aligned}$$

It is important to note that it is not because a rewrite rule is used only in one direction that the theorems need to be proved only in that direction. The syntactic rewriting (specified with the $\vec{=}$ operator in this paper) is not related to the implication operator ($\Rightarrow$). If an expression is to be rewritten to another expression, both should be *equivalent*, or else an error will be introduced. For this reason the proofs were done as equivalences in PVS ($\Leftrightarrow$).

Once expressed in PVS, the interactive proof of a theorem can be undertaken. In the next section, we present the proof results for the rewrite rules presented earlier in the paper, and discuss these and a variety of other experimental results.

V. Proof Results

Using our PVS modeling of the PSL semantics, the rewrite rules presented in Section III were posed as theorems and the proofs were performed interactively by entering inference rules and commands in PVS. The number of commands used to build the proof tree of a given theorem is reported in the tables. These numbers correspond to macro proof commands that are manually entered in the PVS command line. These metrics are meant as a rough indication of proof complexity, and could be improved upon if the actual goal was to craft the shortest possible proofs, which was not our primary intention.

In our experiments, a rule is considered correct even if it does not hold on an empty trace. For those rules that can only be proved on non-empty traces, the theorem is simply modified to exclude empty traces, and then a proof can be obtained in PVS. Proofs of non-equivalence for empty traces are performed separately as they require the modeling of a non-equivalence in the theorem. Neglecting empty traces is usually not a concern in dynamic verification, as simulators, emulators and run-time checkers begin verifying assertions only once the activity has begun in the circuit. In formal verification contexts however, the outcomes marked "Proved*" (with a single asterisk) could be considered as marked "Failed." In some cases, a slight workaround to the rewrite rule can be introduced to correct this issue if need be.

We first summarize the outcome of the proofs for the set of PSL rewrite rules employed in the MBAC checker generator, with further explanations given regarding a few particular proof outcomes. The proofs results for the Cimatti *et al.* rewrite rules follow in Section V-B, and we close this section by reporting on the Singh and Garg cases.

TABLE I
PROOF RESULTS FOR MBAC REWRITE RULES (PROPOSITIONS 1–3)

Rule	Result	# Comm.	Rule	Result	# Comm.
R1	Proved	85	R16	Proved	97
R2	Proved*	49	R17	Proved	169
R3	From Def.	N.A.	R18	Proved	8
R4	Proved	21	R19	Proved	4
R5	Proved**	63	R20	Proved	7
R6	Proved	133	R21	Proved	7
R7	Proved	189	R22	Proved*	402
R8	Proved	768	R23	Proved*	203
R9	Proved**	41	R24	Proved*	14
R10	Proved	971	R25	Proved*	14
R11	Proved	212	R26	Proved*	18
R12	Proved	420	R27	Proved*	15
R13	Proved	191	R28	Proved	7
R14	Proved	2481	R29	Proved	7
R15	Proved	740	R30	Proved	126
			Lemmas	(17)	2017

* With exception of empty trace.

** Using a strong sequence on left-hand side of rule.

A. Proofs of MBAC Rewrite Rules

Table I shows a summary of the proof results for the MBAC rewrite rules appearing in Propositions 1–3. Rewrite rule R3 follows directly from the PSL definitions and does not need to be proved.

All twenty nine non-trivial cases were proved, with only two small modifications to the theorems, indicated by * and **. Rule R2 and rules R22–R27 were proven for non-empty traces, and for rules R5 and R9, strong sequences had to be specified in the left side of the rewrite rules. These issues are discussed further at the end of this section.

As can be seen in the table, proof trees range in size from trivial proofs (under ten commands), to almost 2500 commands. In general, short proofs indicate that a given rewrite rule is very close to the language equivalence used to define the operator in the PSL semantics. For example, rules R24–R29 (next* family) can be seen as specialized cases of more general operators, and are thus relatively quick to prove. General operators such as those in the left sides of rules R14 and R15 (next_event_e) require much more effort.

The proofs are semi-automatic, and require a fair amount of human intuition and experience, especially when a proof is complex (for example, R14). In most cases, specific knowledge of the PSL semantics modeled in a given proof sequent (a step in the proof) is required to direct the next step, as opposed to blindly applying inference rules in hopes of succeeding.

Two cases in Table I deserve special explanations since the rewrite rules actually had to be modified, to adjust for what we believe is a suspicious behavior in the PSL semantics. The original form for rewrite rule R5 (never) can be disproved using the counterexample sequence {a;b;c}; as shown next

$$\text{never } \{a;b;c\} \quad \not\equiv \quad \{[+] : \{a;b;c\}\ \} \text{ |-> false}$$

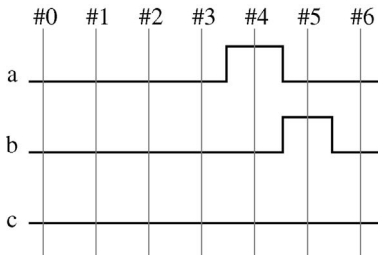

Fig. 4. Waveform for signals a, b, c.

The waveform in Fig. 4 illustrates the values of signals a, b and c. For the trace starting at ♯0 and ending at ♯5, the property never {a;b;c} is false because the sequence as a property {a;b;c} is satisfied on the trace starting at ♯4 and ending at ♯5. But in the same trace, the property {[+] : {a;b;c} } |–> false is satisfied because the sequence {a;b;c} is not tightly satisfied on [♯4, ♯5]; hence there is a contradiction.

The rule that is implemented is in fact the following one:

$$\text{never } r! \stackrel{\rightarrow}{=} \{[+] : r\} \text{ |-> false}$$

According to the standard, a property holds if no bad states have been seen, all future obligations have been met and if on any extension of the trace the property may or may not hold. It is worth noting that the property never {a;b;c} does not hold on the trace [♯0, ♯6] since a bad state has been seen for the trace [♯0, ♯4] padded with $\top$. This semantics problem is mentioned in [21].

With the same waveform and similar arguments, we show that rule R9 is not satisfied; the implemented rule is actually

$$\text{eventually! } r! \stackrel{\rightarrow}{=} \{[+] : r\}!$$

The expression eventually! {a;b;c} holds on sub-trace [♯0, ♯4] and [♯0, ♯5] since {a;b;c} holds on [♯4] and [♯4, ♯5]. Intuitively, we would have wished the property to be pending just as it is on the trace [♯0, ♯6]. In the right-hand side of R9, we show that it does not hold on [♯0, ♯4] nor on [♯0, ♯5] since the sequence must be strongly satisfied. We found no mention of this problem related to the eventually! operator in the literature.

These two rewrite rules, and the successful proofs for the modified versions lead us to believe that the simple subset definition requires a few modifications. These are discussed further in Section VI-B.

The rules R22–R27 regarding the next family are not correct for the case $i = 0$ when the trace is empty. Let us focus on rule R22. Using Definition 5, the left part of R22 is equivalent to p

$$\text{next}[0](p) \equiv p$$

We can now rewrite the right-hand side of R22 using the semantics of the unclocked SEREs and FL from the standard

$$\begin{aligned} & \text{next\_event(true)}[0+1](p) \\ \equiv\ & \text{next\_event(true)}(p) \\ \equiv\ & \text{false until } p \\ \equiv\ & \text{false until! } p \vee \text{always false} \\ \equiv\ & \text{false until! } p \vee \neg\text{eventually! true} \\ \equiv\ & \text{false until! } p \vee \neg(\text{true until true}) \end{aligned}$$

TABLE II
PROOF RESULTS FOR CIMATTI *et al.* REWRITE RULES (PROPOSITION 4)

Rule	Result	# Comm.	Rule	Result	# Comm.
CR1	Proved	92	CR9	Proved	82
CR2	Proved	122	CR10	Proved	25
CR3	Proved	38	CR11	Proved	29
CR4	Proved	57	CR12	Proved	23
CR5	Proved	44	CR13	Proved	96
CR6	Proved	53	CR14	Proved	146
CR7	Proved	44	CR15	Proved	188
CR8	Proved	95	Lemmas	(4)	197

The right-hand side of the disjunction holds on an empty trace, as shown next

$$\begin{aligned} & v \models \neg(\text{true until true}) \\ \equiv\ & \neg(\ \exists k < |v|, v^{k\cdots} \models \text{true}, \quad \wedge \quad \forall j < k, v^{j\cdots} \models \text{true}\ ) \\ \equiv\ & \forall k < |v|, v^{k\cdots} \models \text{false}, \quad \vee \quad \exists j < k, v^{j\cdots} \models \text{false} \end{aligned}$$

If the trace is empty then $\forall k < 0, v^{k\cdots} \models \text{false}$ is true and we can conclude that for any property p, the property next_event(true)[0+1](p) holds on an empty trace. If p is a strong formula, it will not hold on an empty trace.

The same argument applies to rules R23–R27. Thus, it is necessary to restrict rules R22–R27 to $i > 0$, and introduce five additional rewrite rules specific to the case $i = 0$. For instance, for R22 the rule would be

$$\text{next}[0](p) \stackrel{\rightarrow}{=} p$$

Rule R2 (property implication) is not valid for empty traces. Interestingly, defining the semantics of implication with the or operator and Boolean negation (instead of property negation) leads to a successful proof even on empty traces; however, this applies only to the simple subset case where full properties are not used in the antecedent.

B. Proofs of the Cimatti et al. Rewrite Rules

The rewrite rules developed by Cimatti *et al.* [11] are used to simplify various forms of SEREs and suffix implications. Table II shows a summary of the proof results for the set of fifteen rules presented in Proposition 4. All fifteen rewrite rules were proved correct in PVS.

A few lemmas from the proofs performed in the MBAC rewrites were also used in the proofs herein. One new lemma was devised for assisting in the proof of rule CR13. The supplemental lemma expresses the fact that if a SERE in the form r[*] holds on a given trace, it also holds on an arbitrary sub-trace contained within the initial trace. The proof of this lemma was done using strong induction, and required 114 commands to prove separately. As in many proof strategies, preparing intermediate proof constructs (lemmas) can go a long way toward simplifying a proof. This is exemplified in the many lemmas used, particularly in the proofs of the MBAC rewrite rules.

Empty SEREs deserve special attention in some rules. In this paper, the term "empty SERE" describes a SERE that can be tightly satisfied by the empty trace, as in the example {a[*0:4]}, whereas the term "null SERE" describes a SERE that can only be tightly satisfied by the empty trace. A null SERE can be specified in PSL using {[*0]}. The mechanism used to prevent empty SEREs from being modeled in the theorems, as required for rewrite rules CR6, CR8 and CR9, is to ensure that the SERE semantics is not true on any empty trace. It is implemented as follows in PVS (where r is a SERE):

$$\text{not (exists } (t_1\text{: } \mathbb{N})\text{: sem}(r, t_1, t_1 - 1))$$

As such, the above expression is used as an antecedent in an implication, and the semantic equivalence to be proved is entered as the consequent.

A substitute for rule CR6 is shown as rule CR6b

$$\begin{aligned} \text{CR6} &: \{r_1\ ;\ r_2\} \mid\!\!\rightarrow p \stackrel{\rightarrow}{=}^{\dagger} \{r_1\} \mid\!\!\rightarrow \text{next}\ (\{r_2\} \mid\!\!\rightarrow p) \\ \text{CR6b}&: \{r_1\ ;\ r_2\} \mid\!\!\rightarrow p \stackrel{\rightarrow}{=}^{\dagger} \{r_1\} \mid\!\!=\!\!> (\{r_2\} \mid\!\!-\!\!> p) \end{aligned}$$

The alternate rewrite rule is based on non-overlapped suffix implication, and can be used in place of the original rule. It was proved in PVS using roughly the same number of commands as rule CR6. As outlined (†), empty SEREs are disallowed in r_1 and r_2. Computer assisted proofs can also be performed to help ensure correctness when conjecturing on alternate forms of existing rewrite rules, as we have done here.

TABLE III
PROOF RESULTS FOR THE SINGH AND GARG RULES (PROPOSITION 5)

Rule	Result	# Comm.	Rule	Result	# Comm.
DR1	Failed	49	DR6	Proved	217
DR2	Failed	45	DR7	Proved*	203
DR3	Proved*	143	DR8	Failed	101
DR4	Proved	311	DR9	Proved	123
DR5	Failed	185	DR10	Proved* 196	
Lemmas (7)		301			

* With exception of empty trace.

C. Results for the Singh and Garg Cases

To further demonstrate the scope of applicability of the methods proposed in our PSL-PVS framework, we now report on the proof results for the Singh and Garg cases. The set of ten rewrite rules was shown in Proposition 5. Table III shows a summary of the proof results. For each failed case, the number of steps used to prove the non-equivalence of the rule on a counterexample is instead reported.

We now detail the incorrectness proofs for the first two cases, which correspond to the rewrites labeled DR1 and DR2. These two particular cases are not actually introduced as rewrite rules in the original reference [12], but are used to explain the semantics of property operators in the text. We have branded these two cases as rewrite rules in this paper, since a rewrite rule can always be derived from a semantic equivalence, provided the equivalence is correct to begin with.

The approach we adopt essentially resides in finding a counterexample to a given equivalence rule, and then proving the non-equivalence in the case of the counterexample. In the typical equivalence proof, one tries to prove a statement for all possible cases. Often, if the statement cannot be proved, a reason for this becomes apparent during the proof steps and

a counterexample can be derived. The counterexample is then used on a theorem of nonequivalence.

The counterexample for the first equivalence statement consists of a SERE and a trace showing a case where the rule fails to hold. The first statement is related to the eventually! operator, and was presented as rule DR1 in Proposition 5

$$\text{eventually! SERE} \equiv \{\text{true}\} \text{ |-> } \{[*];\text{SERE}\}!$$

This rule has similarities to rewrite rule R9 in Proposition 2. The problem anticipated with this rule, however, is the fact that an empty SERE could be produced in the right side's concatenation with [*]. The counterexample for this rule makes no use of an explicit trace, and can be designed using the SERE {a[*0:4]}. In an abstracted notation, here is how the conjecture is entered in PVS:

$$\begin{aligned} &\forall\ t_0 > 0, T > t_0 - 1 : \\ &\quad v^{t_0..T} \models \{\text{true}\} \text{ |-> } \{[*];\{\text{a[*0:4]}\}\}! \quad \not\Rightarrow \\ &\quad v^{t_0..T} \models \text{eventually! } \{\text{a[*0:4]}\}! \end{aligned}$$

On the trace where a is always false, the property on the left-hand side of the $\Rightarrow$ implication is true, since {a[*0:4]} is satisfied on an empty trace. The property on the right-hand side of $\Rightarrow$ is false since eventually! can be rewritten to: true until {a[*0:4]}!. According to the standard, the second operand of the until operator is satisfied only on a non empty trace. Thus, "a" must be true at least once, hence the contradiction. This shows why in our rule, a fusion with [+] is more appropriate.

Another equivalence in [12] was presented as rewrite rule DR2 in Proposition 5

$$\text{never } SERE \equiv SERE \text{ |-> } \{\text{false}\}$$

This rule was also disproved by counterexample. A counterexample is produced when the SERE starts after the first cycle and does produce a match. In this case, the property on the right-hand side of $\equiv$ will not signal an error (because the SERE does not hold in the initial cycle) while the one on the left will (the SERE evaluation restarts at each cycle).

Rewrite rules DR3, DR7 and DR10 were only proved for non-empty traces. The common reason for this is that in these three cases a strong until! operator is used in the left side of the rule, whereas the right side makes use of suffix implication. A suffix implication can be true on an empty trace since the antecedent is not matched; however, the obligation imposed by the strong until! cannot be met on an empty trace.

The rules DR5 and DR8 were shown incorrect by counter example

$$\begin{aligned} &\text{DR5: next}(r_1 \text{ |-> } r_2) \overset{\rightarrow}{=} \{\text{true};r_1\} \text{ |-> } \{r_2\} \\ &\text{DR8: next}(r_1 \text{ |-> } r_2!) \overset{\rightarrow}{=} \{\text{true};r_1\} \text{ |-> } \{r_2\}! \end{aligned}$$

The following SEREs, when used in the rewrite rules, can be used to prove that the equivalence does not hold on any trace of length two

$$r_1 = \text{empty\_S}, \quad r_2 = \text{false\_S}$$

It is interesting to note, however, that if an empty SERE is disallowed in r_1, as is done in some Cimatti *et al.* rules, the DR5 and DR8 rules can be proved.

TABLE IV
PROPOSED MODIFICATIONS TO SIMPLE SUBSET GUIDELINES

Item[1]	Revised guideline
2	The operand of a never operator is a Boolean or a strong Sequence
3	The operand of an eventually! operator is a Boolean or a strong Sequence
8	The right-hand side operand of an overlapping until* operator is a Boolean
10	The operand of next_e* is a Boolean or a sequence
11	The FL Property operand of next_event_e* is a Boolean or a sequence

[1]Itemized list in Section 4.4.4 of [16].

VI. REVISED SIMPLE SUBSET GUIDELINES

In this section, we first present our formulation of the formal definitions for finite behaviors in PSL, and show how to interpret the satisfaction criteria for finite traces. We also proceed to outline our proposal for simple subset guidelines.

A. Some Formal Definitions for Finite Behaviors

The PSL language reference defines four levels of satisfaction of a property: holds strongly, holds, pending, fails, but gives no formal definition. These satisfaction levels are defined using some intermediate definitions that we have formalized to ease the understanding of our discussion on the unsuccessful rewrite rules.

Let φ be a property and v be a trace.

1) No bad states have been seen. This is defined by
$$\forall j \leq |v|, v^{0..j}\top^{\omega} \models \varphi$$
2) All future obligations have been met. The definition is
$$v \models \varphi$$
3) The property will hold on any extension of the path. In this case, it must hold on the trace padded with extra $\perp$'s
$$v\perp^{\omega} \models \varphi$$

Combinations of these intermediate definitions define the four satisfaction levels mentioned above.

B. Syntactic Definition of the Simple Subset

We present our view of what the simple subset guidelines should be, in the light of the findings arising from our proof efforts. Of the eleven conditions outlined in the simple subset guidelines, in Section 4.4.4 of [16], five require modifications. Table IV shows how we believe these conditions should appear.

To summarize, the modifications for items 2 and 3 are a consequence of the PVS proofs we performed in this paper, as discussed in Section V-A. Consequently to the proofs performed, the word "strongly" should be added to further characterize the sequences that can be used in these two operators. The modification to item 8 is part of the simple subset issues of the working group [20], and in this paper was

proved to be coherent with our run-time semantics and rewrite rules. The original formulation of this guideline required that both operands be of type Boolean.

The modifications to items 10 and 11 are a consequence of analyzing the rewrite rules corresponding to these operators. Here, the original formulation required that only a Boolean be used in the given operand. The proposed modification allows sequences to also be used while still preserving a certain amount of temporal order in the property, enough for it to be considered simulation-friendly. To illustrate how we obtain the relaxed conditions of allowing sequences, consider the rewrite rule for the next_event_e operator (the strong version and the two next_e cases are similar)

$$\text{R14: next\_event\_e}(b_1)[k{:}l](b_2) \stackrel{\rightarrow}{=} \{b_1[{-}{>}k{:}l] : b_2\}$$

In the right-hand side of the above rule, the second argument of the fusion (:) could actually be a sequence instead of the Boolean b_2. Hence, the less constrained condition for the simple subset could be as follows (where r is a SERE):

$$\text{next\_event\_e}(b)[k{:}l](r) \stackrel{\rightarrow}{=} \{b[{-}{>}k{:}l] : r\}$$

The reasoning behind our simple subset version is that there should be a certain amount of consistency in the guidelines such that rules are well founded no matter what form one chooses to express a property. To make a case, consider again the last rewrite rule. It would seem illogical that the form

$$\text{next\_event\_e}(b)[k{:}l](r)$$

be excluded from the simple subset while the equivalent form

$$\{b[{-}{>}k{:}l] : r\}$$

is easily part of the simple subset. We believe that the rewrite rules provide a key mechanism to formally define the simple subset guidelines and help provide a closed subset that is consistent.

VII. Conclusion and Continuing Work

In this paper, we have shown how automated theorem proving can be used to effectively prove the assertion rewrite rules and equivalences in modern hardware assertion languages such as PSL. Due to the complexity of the language and the counterintuitive nature of some of the rewrite rules, we have shown that some of the published rewrites were actually incorrect. Of the more than 50 rewrite rules from various sources retained for experiments in this paper, two were not provable because of language semantics issues, nine were shown not to hold on empty traces only, and four were shown to be false.

As witnessed in our proof results, we also showed how certain simple subset guidelines must be changed in order to create behaviors that are better suited to dynamic verification with PSL. We have formally justified the guidelines for writing a simulation friendly PSL. In our view, the simple subset is the set of properties that can be rewritten to a property compliant with our Definition 6 along with one or more of the proved correct rewrite rules.

Our method is easily applicable to SVA semantics as well. Given the relative similarities of SVA and PSL [22], proving theorems about SVA semantics and rewrite rules is quite similar. New extensions will be required in our PVS framework to handle local variable assignments in SVA sequences [23]. A noteworthy challenge also remains to prove theorems about multi-clock assertions, that is, single assertions containing subexpressions that are synchronized to different clock rates.

Acknowledgment

The authors would like to thank the anonymous reviewers in particular for a meticulously detailed analysis, which was instrumental to improving the paper.

References

[1] H. Foster, A. Krolnik, and D. Lacey, *Assertion-Based Design*, 2nd ed. Norwell, MA: Kluwer, 2004.

[2] Y. Abarbanel, I. Beer, L. Glushovsky, S. Keidar, and Y. Wolfsthal, "FoCs: Automatic generation of simulation checkers from formal specifications," in *Proc. 12th Int. Conf. CAV*, 2000, pp. 538–542.

[3] K. Morin-Allory and D. Borrione, "Proven correct monitors from PSL specifications," in *Proc. 2006 Conf. DATE*, 2006, pp. 1246–1251.

[4] M. Boulé and Z. Zilic, "Efficient automata-based assertion-checker synthesis of PSL properties," in *Proc. IEEE Int. HLDVT Workshop*, 2006, pp. 69–76.

[5] R. Armoni, D. Korchemny, A. Tiemeyer, M. Y. Vardi, and Y. Zbar, "Deterministic dynamic monitors for linear-time assertions," in *Proc. Workshop FATES/RV*, LNCS 4262. 2006, pp. 163–177.

[6] M. Boulé, J. Chenard, and Z. Zilic, "Debug enhancements in assertion-checker generation," *IET Comput. Digital Tech.: Special Issue Silicon Debug Diagnosis*, vol. 1, no. 6, pp. 669–677, 2007.

[7] M. Boulé and Z. Zilic, *Generating Hardware Assertion Checkers: For Hardware Verification, Emulation, Post-Fabrication Debugging and On-Line Monitoring*. Berlin, Germany: Springer, 2008.

[8] Y. Oddos, M. Boulé, K. Morin-Allory, D. Borrione, and Z. Zilic, "MYGEN: Automata-based on-line test generator for assertion-based verification," in *Proc. ACM Int. GLVLSI*, 2009, pp. 75–80.

[9] K. Morin-Allory and D. Borrione, "A proof of correctness for the construction of property monitors," in *Proc. IEEE Int. HLDVT Workshop*, 2005, pp. 237–244.

[10] M. Boulé and Z. Zilic, "Automata-based assertion-checker synthesis of PSL properties," *ACM Trans. Design Automat. Electron. Syst.*, vol. 13, no. 1, p. 4, Jan. 2008.

[11] A. Cimatti, M. Roveri, and S. Tonetta, "Symbolic compilation of PSL," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 27, no. 10, pp. 1737–1750, Oct. 2008.

[12] V. Singh and T. Garg, "Transformation of simple subset of PSL into SERE implication formulas for verification with model checking and simulation engines using semantic preserving rewrite rules," U.S. Patent Applicat. 20 060 136 879, Jun. 22, 2006.

[13] M. Gordon, J. Hurd, and K. Slind, "Executing the formal semantics of the Accellera property specification language by mechanized theorem proving," in *Proc. CHARME*, LNCS 2860. Oct. 2003, pp. 200-215.

[14] K. Claessen and J. Mårtensson, "An operational semantics for weak PSL," in *Proc. FMCAD*, LNCS 3312. Nov. 2004, pp. 337–351.

[15] K. Ng, A. Hu, and J. Yang, "Generating monitor circuits for simulation-friendly GSTE assertion graphs," in *Proc. 22rd IEEE ICCD*, 2004, pp. 288–492.

[16] IEEE Std. 1850–2005, *IEEE Standard for Property Specification Language (PSL)*. New York: IEEE Press, 2005.

[17] K. Morin-Allory, M. Boulé, D. Borrione, and Z. Zilic, "Proving and disproving assertion rewrite rules with automated theorem provers," in *Proc. IEEE Int. HLDVT Workshop*, 2008, pp. 56–63.

[18] *PVS Prover Guide*, Comput. Sci. Laboratory, SRI Int., Menlo Park, CA, Dec. 2001.

[19] C. Eisner and D. Fisman, *A Practical Introduction to PSL*. New York: Springer, 2006.

[20] IEEE Std. 1850-200x Working Group, "Simple Subset Issue #99, Group E.1," Issues to be Addressed in IEEE 1850-200x PSL, 2006.

[21] IEEE Std. 1850-200x Working Group, "Unaddressed Issue #146," Issues to be Addressed in IEEE 1850-200x PSL, 2006.

[22] J. Havlicek, D. Fisman, C. Eisner, and E. Marschner. (2003). *Mapping SVA to PSL* [Online]. Available: http://www.eda.org/vfv/docs/mapping.pdf

[23] J. Long and A. Seawright, "Synthesizing SVA local variables for formal verification," in *Proc. 44th DAC*, 2007, pp. 75–80.

Katell Morin-Allory received the M.S. and Ph.D. degrees from the University of Rennes 1, Rennes, France, in 2002 and 2004.

She is an Associate Professor with Grenoble Institute of Technology (Grenoble INP), Grenoble, France. Her research interests are in the areas of hardware design languages, simulation, formal verification, synthesis, and asynchronous designs.

Dr. Morin-Allory is a member of the TIMA Laboratory.

Marc Boulé received the M.Eng. and Ph.D. degrees in electrical engineering from McGill University, Montreal, QC, Canada, in 2002 and 2008.

He is a full time Instructor with the École de Technologie Supérieure, Montreal. Along with Z. Zilic, he has co-authored the book titled *Generating Hardware Assertion Checkers*, which showcases core techniques for producing circuit-level checkers from properties written in high-level assertion languages. His current research interests include assertion-based verification, digital design, automated theorem proving, and field-programmable gate arrays.

Dr. Boulé received an Honorary Mention in the Natural Sciences and Engineering Research Council of Canada Innovation Challenge Awards for his Ph.D. research project in 2008.

Dominique Borrione received the M.S., Doctorat d'Etat, and Ph.D. degrees from the University of Grenoble, Grenoble, France.

She is a Full Professor with the Polytech Department, Joseph Fourier University, Grenoble. She is the Director of the Techniques de l'Informatique et de la Microélectronique pour l'Architecture des systèmes intégrés Laboratory, University of Grenoble (CNRS-GrenobleINP-UJF), Grenoble, France, and Former Director of the ARTEMIS Laboratory, University of Grenoble. She has co-authored over 130 refereed articles and book chapters. Her current research interests include hardware design languages, specification, CAD, simulation, formal verification, and synthesis.

Dr. Borrione has served in many working groups and yearly conference committees, and was the Program Chair of CHARME'86 and '05, CHDL'91, DATE'99, and FDL'09. She received the distinguished International Federation for Information Processing Silver Core in 1986.

Zeljko Zilic (M'97–SM'07) received the Ph.D. and M.S. degrees in electrical and computer engineering from the University of Toronto, Toronto, ON, Canada.

From 1996 to 1997, he was with Lucent Microelectronics, Allentown, PA. Since 1998, he has been with McGill University, Montreal, QC, Canada, where he is currently an Associate Professor with the Department of Electrical Engineering. His current research interests lie in the area of design, test and the design for quality of integrated systems. He has published over 200 publications, and has led or participated in program committees for numerous conferences such as HLDVT, International Symposium on FPGAs, ITC, Silicon Debug and Diagnosis, MWSCAS, NEWCAS and AQTR conferences, ICECS, and IOST3. He is a senior member of ACM.