Proving and Disproving Assertion Rewrite Rules with Automated Theorem Provers

Katell Morin-Allory[†], Marc Boulé[‡], Dominique Borrione[†] and Zeljko Zilic[‡]
[†]TIMA Laboratory, 46 avenue Félix Viallet 38031 Grenoble Cedex, France
{katell.morin-allory, dominique.borrione}@imag.fr
[‡]McGill University, 3480 University Street, Montreal, Quebec, Canada
marc.boule@elf.mcgill.ca, zeljko.zilic@mcgill.ca

Abstract— **Modern assertion languages, such as PSL and SVA, include many constructs that are best handled by rewriting to a small set of base cases. Since previous rewrite attempts 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, workable procedures for proving the correctness of these rules must be established. In this paper, we outline the methodology for computer-assisted proofs of a set of previously published rewrite rules for PSL properties. We show how to express PSL's syntax and semantics in the PVS theorem prover, and proceed to prove the correctness of a set of thirty rewrite rules. In doing so, we also demonstrate how to circumvent issues with PSL semantics regarding the never and eventually! operators.**

I. Introduction

The use of assertions in hardware [1] has increased rapidly in the past decade, and has helped to deal with the increased challenges in hardware verification. With the standardization of the two main assertion languages, namely PSL (Property Specification Language) and SVA (SystemVerilog Assertions), the industry now has a formally defined and industrially robust means of specifying correctness properties in hardware designs. Although the main goal of assertions is to help the verification stage, assertions also serve as a formal documentation mechanism useful for both designers and verification teams. When converted to checkers [2], [3], the assertion verification capability can be used in hardware emulation and post-fabrication debugging [4] to augment visibility and assist in locating errors.

Recently, rewrite rules have been used in the MBAC checker generator [5] to help implement the various forms of property operators in PSL. Although this checker generator is based on automata for expressing properties, rewrite rules play a key role. In a parallel effort, the HORUS checker generator has had its modular approach to property construction formally proven by automated theorem proving [2], [6]. This paper presents a synergy of research done by two teams devoted to checker generators. While we deal mainly with proving the correctness of MBAC's rewrite rules 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. Since both tools find their main use in dynamic property verification through assertion checkers, the underlying context throughout this paper is that of dynamic property checking. Yet, the method is generic and applicable also to rewrite rules intended for static property checking.

PSL defines a set of restrictions to the language 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 [7]. In the checkers developed for GSTE assertion graphs, certain constructs are avoided or restricted in scope, for hardware implementation reasons.

The PSL simple subset guidelines (Section 4.4.4 in [8]) 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. In this paper, the PSL simple subset is used (PSL-SS), as described in the next section.

The main contribution of this paper is a formally proven set of rules to rewrite the large variety of PSL temporal operators into a more elementary syntactical PSL subset. The proofs are founded on a higher-order semantic definition of PSL in the logic of PVS [9] and on the use of the PVS proof assistant to mechanize the formal reasoning. One of the consequences of this work is a formally justified revision proposal for the PSL simple subset.

Previous work along these lines are few. We found the work of M. Gordon particularly inspiring, in which he performed a "deep embedding" of PSL in the 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 [10]. The direct execution of the PSL semantics in HOL complies to the language definition by construction, but the resulting observers are too inefficient to be used in large designs. In contrast, we formally prove the principles on which production quality checkers could be generated for all the user-friendly operators of PSL, on top of a proven correct implementation for a kernel of PSL primitives. Other work by Claessen and Martensson proposed an operational semantic definition guided by the structure of weak PSL formulas [11], and exhibited some inconsistencies in the interpretation of a special class of regular expressions. The main purpose of that work was to investigate alternative semantic definitions to the trace semantics, as a way to fix problems with the early versions of PSL. If and how this was useful to guide an implementation of property checkers has not been published.

For the paper to be self-contained, Section II provides the kernel semantics of PSL, adapted from [8]. Readers already familiar with PSL and its simple subset may jump to Section III, that discusses the set of rewrite rules to be proved. Section IV shows how to perform the proofs, and in Section V these and other proof results are detailed.

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. Standardized by the IEEE in 2005, the PSL language has a formally defined syntax and semantics, and is documented in the PSL Language Reference Manual (LRM) [8]. Other sources offer more pragmatic treatments of PSL and can be better suited to grasp the intricacies of such languages [12].

The Verilog flavor of PSL is used throughout the paper; however, the proofs and techniques are not restricted to PSL since many concepts can apply to the proof of SVA rewrite rules for example. In the presentation made here, the simple subset guidelines [8] are applied.

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 are 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 $v^{i..j}$, 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 $v^{i..}$, 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):

(i) $l \,|\!\models p \quad \Leftrightarrow \quad p \in l$
(ii) $l \,|\!\models \,!b \quad \Leftrightarrow \quad l \,|\!\not\models b$
(iii) $l \,|\!\models b_1 \,\&\, b_2 \quad \Leftrightarrow \quad l \,|\!\models b_1$ and $l \,|\!\models b_2$

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

$$l \,|\!\models b_1 \text{ (i.e. } \langle \mathsf{a}, \mathsf{c} \rangle \text{ satisfies } \mathsf{c})$$

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

In this paper the Booleans true and false are used to 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. The two special letters $\top$ and $\bot$ are 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.

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 [8].

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

- b
- $\{r\}$
- $r_1 \,;\, r_2$
- $r_1 : r_2$
- $r_1 \,|\, r_2$
- 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 $l v_2 \models r_2$
- $v \models r_1 | r_2$ 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[+] $\overset{\text{def}}{=}$ $r\,;\,r$[*]
- r[*k] $\overset{\text{def}}{=}$ $r\,;\,r\,;\,\ldots\,;\,r$ (k times)
- r[*i:j] $\overset{\text{def}}{=}$ r[*i] | … | r[*j]
- b[–>] $\overset{\text{def}}{=}$ {(!b)[*] ; b}
- b[–>k] $\overset{\text{def}}{=}$ {b[–>]}[*k]
- b[–>k: l] $\overset{\text{def}}{=}$ {b[–>]}[*k:l]

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 main property operators defined in PSL, with the simple subset guidelines applied [8]. However, according to the current PSL Issues List (item #99, simple subset issues), the overlapping until operator can allow a full property in the left-side argument, as opposed to only a Boolean as originally indicated in the simple subset guidelines.

Definition 4: If b is a Boolean expression, r is a SERE and p is a property, then PSL *Foundation Language (FL) properties* are defined as follows in the simple subset.

- b
- !b
- b_1 <–> b_2
- (p)
- p abort b
- p_1 && p_2
- r
- r!
- r |–> p

The operators shown above are the base cases that have specific automata implementations in [5]; these do not have rewrite rules and are based on the semantics defined in Appendix B in [8]. In the simple subset, negation and equivalence (<–>) apply only to Booleans, and are therefore implicitly handled in the semantics of Booleans. The semantics of (p) is the same as the semantics of p since the parentheses are used only for grouping. The semantics of the six remaining base property operators is defined next.

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, or stated otherwise, that trace v satisfies property p:

- $v \models b$ iff $|v| = 0$ or $v^0 \Vdash b$
- $v \models p_1$&&p_2 iff $v \models p_1$ and $v \models p_2$
- $v \models r$|–>p iff $\forall j < |v|$ s.t. $\overline{v}^{0..j} \equiv\!\!\!| r$, $v^{j..} \models p$
- $v \models r!$ iff $\exists j < |v|$ s.t. $v^{0..j} \equiv\!\!\!| r$
- $v \models r$ iff $\forall j < |v|$, $v^{0..j}\top^{\omega} \models r!$
- $v \models p$ abort b iff either $v \models p$ or $\exists j < |v|$ s.t. $v^j \Vdash b$ and $v^{0..j-1}\top^{\omega} \models p$

The syntax of the remaining property operators is not shown in Definition 4 since these operators are presented in the next section. The additional property operators in PSL appear as the left side in the rewrite rules. For these operators, the semantics will be explained informally. Once the rules will be proven correct, the semantics can be determined by inductively applying the rewrite rules to reach the base cases, where the semantics is shown above.

III. Rewrite Rules

In many languages such as PSL, only a few base operators are actually needed to define the language, and supplemental operators are added only as a syntactical convenience for writing expressions. For example, in *pure* LTL (Linear Temporal Logic), 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. PSL properties are based on LTL, with the addition of a wide array of sugaring operators. Most of these operators are defined as language equivalencies in the PSL specification [8]. Because of the restrictions imposed by the simple subset, these equivalencies can generally not be used as rewrite rules [5]. For this reason, a set of rewrite rules particularly suited to dynamic verification was developed and used in the MBAC checker generator. 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. Rewrite rules for PSL's simple subset have also been developed by Cadence [13] for an early version of PSL.

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 $\overset{\rightarrow}{\equiv}$ symbol is used to show that both side of the rule are in fact equivalent (this is confirmed 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 subsequently rewritten using other rules, until no more rewrites apply and either sequences, Boolean expressions or base cases 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.

The set of thirty rewrite rules for PSL presented below were developed for the MBAC checker generator [3], [5]. They make use of the syntactic sugaring already defined in the standard for SERE's and constitute an alternative definition to the "additional FL operators" (section B.4.2 of [8]). The rewrite rules are grouped into three themes, according to a common characteristic they share.

Proposition 1: Rewrite Rules Based on Property Implication.

R1: b || p $\overset{\rightarrow}{\equiv}$ {!b} |–> p
R2: b –> p $\overset{\rightarrow}{\equiv}$ {b} |–> p
R3: r |=> p $\overset{\rightarrow}{\equiv}$ {r ; true} |–> p
R4: always p $\overset{\rightarrow}{\equiv}$ {[+]} |–> p
R5: never r $\overset{\rightarrow}{\equiv}$ {[+] : r} |–> false
R6: p until b $\overset{\rightarrow}{\equiv}$ {(!b)[+]} |–> p
R7: p until_ b $\overset{\rightarrow}{\equiv}$ {{(!b)[+]}|{b[–>]}} |–> p
R8: next_event_a(b)[k:l](p) $\overset{\rightarrow}{\equiv}$ {b[–>k:l]} |–> p

The R1 rule is based on the fact that if the Boolean expression is false, then the property must be true; otherwise the 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 [8]. 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 [5], 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: Rewrite Rules Based on Sequences.

R9: eventually! r $\overset{\rightarrow}{=}$ {[+] : r}!
R10: b_1 before b_2 $\overset{\rightarrow}{=}$ {(!b_1&!b_2)[*] ; (b_1&!b_2)}
R11: b_1 before! b_2 $\overset{\rightarrow}{=}$ {(!b_1&!b_2)[*] ; (b_1&!b_2)}!
R12: b_1 before_ b_2 $\overset{\rightarrow}{=}$ {(!b_1&!b_2)[*] ; b_1}
R13: b_1 before!_ b_2 $\overset{\rightarrow}{=}$ {(!b_1&!b_2)[*] ; b_1}!
R14: next_event_e(b_1)[k:l](b_2) $\overset{\rightarrow}{=}$ {b_1[->k:l] : b_2}
R15: next_event_e!(b_1)[k:l](b_2) $\overset{\rightarrow}{=}$ {b_1[->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 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: Rewrite Rules Based on Property Variations.

R16: p until! b $\overset{\rightarrow}{=}$ (p until b) && ({b[->]}!)
R17: p until!_ b $\overset{\rightarrow}{=}$ (p until_ b) && ({b[->]}!)
R18: next p $\overset{\rightarrow}{=}$ next[1](p)
R19: next! p $\overset{\rightarrow}{=}$ next![1](p)
R20: next_event(b)(p) $\overset{\rightarrow}{=}$ next_event(b)[1](p)
R21: next_event!(b)(p) $\overset{\rightarrow}{=}$ next_event!(b)[1](p)
R22: next[i](p) $\overset{\rightarrow}{=}$ next_event(true)[$i+1$](p)
R23: next![i](p) $\overset{\rightarrow}{=}$ next_event!(true)[$i+1$](p)
R24: next_a[i:j](p) $\overset{\rightarrow}{=}$ next_event_a(true)[$i+1:j+1$](p)
R25: next_a![i:j](p) $\overset{\rightarrow}{=}$ next_event_a!(true)[$i+1:j+1$](p)
R26: next_e[i:j](b) $\overset{\rightarrow}{=}$ next_event_e(true)[$i+1:j+1$](b)
R27: next_e![i:j](b) $\overset{\rightarrow}{=}$ next_event_e!(true)[$i+1:j+1$](b)
R28: next_event(b)[k](p) $\overset{\rightarrow}{=}$ next_event_a(b)[k:k](p)
R29: next_event!(b)[k](p) $\overset{\rightarrow}{=}$ next_event_a!(b)[k:k](p)
R30: next_event_a!(b)[k:l](p) $\overset{\rightarrow}{=}$
next_event_a(b)[k:l](p) && {b[->l]}!

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.

Although some rules may appear intuitive, the whole set is presented for completeness. For the rules that are not so intuitive, the computer assisted proofs of correctness help to confirm that they are well founded. With complete proofs, these rules can be used with even greater confidence, in any tool that supports PSL in dynamic verification.

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 are 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 and foundation language (FL) properties. Then we modeled the semantics of each data type by a polymorphic function Sem. This modeling mimics the semantics definition from the PSL 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 data types 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.

```
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

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`, `sig`, `and_bool`, `or_bool`, `not_bool`. The constructors `true_S` and `false_S` take no argument, they represent the constants 1'b1 and 1'b0 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.

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 1'b1.

```
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

A trace is a mapping from $\mathbb{N}$ to `Trace_PSL`. We denote $\mathcal{T}$ 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:

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

The mapping evaluates $v^t \models b$, and its definition in PVS is given in Figure 3. If the value of v^t is $\top$ ($\bot$), then $\text{Sem}(b,t,v)$ is true (false). Otherwise it depends on the semantics of b. If b is an atomic proposition `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. Figure 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.

```
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

In the sequel, for reasons of space only the mathematical semantic definition is given. The reader should be convinced that its translation to the input format of PVS is direct.

The semantics of SEREs is modeled by a mapping

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

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 less than t_0. The translation from the LRM to the modeling is straightforward and we illustrate it using an example. The value of the semantics of s[*] is given by the disjunction of the semantics of [*0] and the semantics of s[+].

$$\begin{array}{rl} & \text{Sem}(s\texttt{[*]}, t_0, T, v) = \\ & \quad \text{Sem}(\texttt{[*0]}, t_0, T, v) \\ \vee & \exists t_1 \in \mathbb{N} : t_0 \leq t_1 \leq T \wedge \text{Sem}(s\texttt{[+]}, t_1, T, v) \end{array}$$

The semantics of FL properties is modeled by a mapping Sem:

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

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 in the following example. The semantics of f_1 until! f_2 is taken directly from the LRM and is defined by:

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

C. Theorem Modeling

For each rewrite rule, a theorem defines the equivalence in PVS. The modeling for rule R16 (strong until) is shown below in (1), as an example. The equivalence is proved regardless of the length of the trace, and for traces that are fully defined (with no $\top$ nor $\bot$).

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

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

V. Proof Results

First we summarize the proof results for the rewrite rules presented in this paper. A few formal definitions are then stated to show how the satisfaction criteria for finite traces are understood. In a third subsection, further explanations are given regarding the proof results. We then briefly report two instances of faulty language equivalence found in the literature, and conclude the section with our proposal for simple subset guidelines.

TABLE I
Proof Results

Rule	Result	# Comm.	Rule	Result	# Comm.
R1	Proved	85	R16	Proved	97
R2	From Def.	N.A.	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

*: using a strong sequence on LHS of rule
**: with exception of empty trace when $i = 0$

A. Proof Results

Using the PVS modeling of the PSL semantics discussed previously, the rewrite rules presented in Section III were modeled as PVS theorems and the proofs were performed interactively by entering inference rules and commands in the PVS interface. The rules for rewrites R2 and R3 follow directly from the PSL definitions and do not need to be proved.

Table I shows a summary of the proof results. All twenty eight of the non-trivial cases were proved, with only two small modifications to the theorems, indicated by * and **. For rules R5 and R9, strong sequences had to be specified in the left side of the rewrite rules, as will be explained further. Also, for the set of rules R22 to R27, the proofs were not successful for empty traces when the repetition count i is equal to zero.

The number of commands used to build the proof trees is also reported in the table. These correspond to macro proof commands that are actually entered in the PVS command line. Proof trees range in size from trivial proofs (under ten commands), to almost 2500 commands. The proofs are semi-automatic, and require a fair amount of human intuition and experience, especially when a proof is complex (ex.: R14). Short proofs indicate that a given rewrite rule is very close to the language equivalence used to define the operator in the PSL semantics.

B. Some Formal Definitions for Finite Behaviors

The PSL manual 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:

- No bad states have been seen. This is defined by

$$\forall j \leq |v|, v^{0..j} \top^{\omega} \models \varphi$$

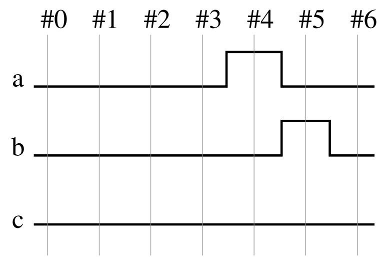

Fig. 4. A waveform for signals a, b, c

- All future obligations have been met. The definition is

$$v \models \varphi$$

- 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.

C. Discussion on the Rewrite Rules

Two cases in Table I deserve special explanations since the rewrite rules actually had to be modified, to correct 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.

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

The waveform in Figure 4 illustrates the values of signals a, b and c. For the trace starting at $\sharp0$ and ending at $\sharp5$, the property never {a;b;c} is false because the sequence as a property {a;b;c} is satisfied on the trace starting at $\sharp4$ and ending at $\sharp5$. But in the same trace, the property {[+] : {a;b;c} } |–> false is satisfied because the sequence {a;b;c} is not tightly satisfied on $[\sharp4, \sharp5]$; hence there is a contradiction.

The rule that is implemented is in fact the following one

$$\text{never } r! \quad \overrightarrow{\equiv} \quad \text{\{[+] : } r\text{\} |-> false}$$

According to the LRM, 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 $[\sharp0, \sharp6]$ since a bad state has been seen for the trace $[\sharp0, \sharp4]$ padded with $\top$. This semantics problem is mentioned in [14].

With the same waveform and similar arguments, we show that Rule 9 is not satisfied; the implemented rule is actually:

$$\text{eventually! } r! \quad \overrightarrow{\equiv} \quad \text{\{[+] : } r\text{\}!}$$

The expression eventually! {a;b;c} holds on sub-trace $[\sharp0, \sharp4]$ and $[\sharp0, \sharp5]$ since {a;b;c} holds on $[\sharp4]$ and $[\sharp4, \sharp5]$. Intuitively, we would have wished the property to be pending just as it is on the trace $[\sharp0, \sharp6]$. In the right hand side of R9, we show that it does not hold on $[\sharp0, \sharp4]$ nor on $[\sharp0, \sharp5]$ 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 Subsection V-E.

The rules R22 to R27 regarding the next family are not correct for the case $i = 0$ when the trace is empty. Although this is not a concern in dynamic verification, it could be important in static property checking. Let us focus on rule R22 for the case $i = 0$. Using the semantics definition, the left part of R22 is equivalent to p as shown below:

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

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

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

The right hand side of the conjunction holds on an empty trace, as shown next:

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

If the trace is empty then $\forall k < 0, v^{k\cdots} \models$ false is obviously 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, and the rule R22 is not true on an empty trace.

New rules can be added to take into account the case $i = 0$ to simplify the property to its operand: next(0)[p] could simply be rewritten to p, and so forth for the other five rules.

D. Disproving Other Rules in the Literature

To demonstrate the scope of applicability of the methods proposed in the PSL-PVS framework, we now prove the incorrectness of two statements appearing in [13]. The statements are not claims as such, but are used to explain the semantics of property operators in the text, and are presented as equivalence rules between two types of expressions.

The proof approach we adopted essentially resides in finding a counterexample to a given equivalence rule, and then in proving the non-equivalence in the case of the counterexample. The counterexample consists of a SERE and a trace showing a case where the rule fails to hold. The first equivalence is related to the eventually! operator, and is very similar to rewrite rule R9 in Proposition 2.

$$\text{eventually! } SERE \quad \equiv \quad \text{\{1'b1\} |-> \{[*];}SERE\text{\}!}$$

The problem anticipated with this rule is the fact that the empty SERE could be produced in the right side's concatenation with [*]. The counterexample for this rule can be designed using the SERE {a[*0:4]}. In an abstracted notation, here is how the conjecture is entered in PVS:

$$\forall\ t_0 > 0, T > t_0 - 1 :$$
$$v^{t_0..T} \models \text{\{1'b1\} |-> \{[*];\{a[*0:4]\}\}!} \quad \not\Rightarrow$$
$$v^{t_0..T} \models \text{eventually! \{a[*0:4]\}!}$$

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: true until {a[*0:4]}!. According to the LRM, the second operand of 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 [13] was proven not to hold:

$$\text{never } SERE \quad \equiv \quad SERE \text{ |-> \{1'b0\}}$$

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 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)

E. A Syntactic Definition of the Simple Subset

We close this section with our view of what the simple subset guidelines should be, in light of our research and the findings contained in this paper. Of the eleven conditions outlined in the simple subset guidelines, in Section 4.4.4 of the IEEE 1850-2005 PSL specification, five require modifications. Here is how we believe these conditions should appear:

- 2^{nd} item: "The operand of a never operator is a Boolean or a **strong** Sequence"
- 3^{rd} item: "The operand of an eventually! operator is a Boolean or a **strong** Sequence"
- 8^{th} item: "The **right-hand side operand** of an overlapping until* operator is a Boolean"
- 10^{th} item: "The operand of next_e* is a Boolean **or a sequence**"
- 11^{th} item: "The FL Property operand of next_event_e* is a Boolean **or a sequence**"

To summarize, the modifications for items 2 and 3 are a consequence of the PVS proofs we performed in this paper, as discussed previously. The modification to item 8 is part of the simple subset issues of the working group [15]. The modifications to items 10 and 11 are a consequence of analyzing the rewrite rules corresponding to these operators. 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):

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

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

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

VI. Conclusions and Continuing Work

In this paper we have shown how automated theorem proving can be used to effectively prove and disprove assertion rewrite rules and equivalencies in hardware assertion languages such as PSL. Furthermore, as witnessed in our proof results, we show 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. Our definition is straightforward: the simple subset is the set of properties that can be rewritten to a property compliant to our Definition 4 along with one or more of the proven correct rewrite rules.

Future work also includes modeling automata in PVS and proving the correctness of the automata algorithms and the implementation of the base cases [5] in the MBAC checker generator. If this can be performed, then the two fundamental methods used in the checker generator will have been proven, namely automata algorithms and rewrite rules.

References

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

[2] K. Morin-Allory and D. Borrione, "Proven Correct Monitors from PSL Specifications," in *Proceedings of the 2006 Conference on Design Automation and Test in Europe (DATE'06)*, 2006, pp. 1246–1251.

[3] M. Boulé and Z. Zilic, "Efficient Automata-Based Assertion-Checker Synthesis of PSL Properties," in *Proceedings of the 2006 IEEE International High Level Design Validation and Test Workshop (HLDVT'06)*, 2006, pp. 69–76.

[4] M. Boulé, J. Chenard, and Z. Zilic, "Debug Enhancements in Assertion-Checker Generation," *IET Computers and Digital Techniques – Special Issue on Silicon Debug and Diagnosis*, vol. 1, no. 6, pp. 669–677, 2007.

[5] M. Boulé and Z. Zilic, "Automata-Based Assertion-Checker Synthesis of PSL Properties," *ACM Transactions on Design Automation of Electronic Systems (ACM-TODAES)*, vol. 13, no. 1, p. Article 4, January 2008.

[6] K. Morin-Allory and D. Borrione, "A Proof of Correctness for the Construction of Property Monitors," in *Proceedings of the 2005 IEEE International High Level Design Validation and Test Workshop (HLDVT'05)*, 2005, pp. 237–244.

[7] K. Ng, A. Hu, and J. Yang, "Generating Monitor Circuits for Simulation-Friendly GSTE Assertion Graphs," in *Proc. of the 22rd IEEE International Conference on Computer Design (ICCD'04)*, 2004, pp. 288–492.

[8] IEEE Std. 1850-2005, *IEEE Standard for Property Specification Language (PSL).* New York, NY, USA: Institute of Electrical and Electronic Engineers, Inc., 2005.

[9] N. Shankar, S. Owre, J. Rushby, and D. Stringer-Calvert, *PVS Prover Guide*, Computer Science Laboratory, SRI International, Menlo Park, CA, Dec. 2001.

[10] M. Gordon, J. Hurd, and K. Slind, "Executing the formal semantics of the Accellera Property Specification Language by mechanised theorem proving," in *Correct Hardware Design and Verification Methods (CHARME'03)*, D. Geist and E. Tronci, Eds., vol. 2860. LNCS , Springer, Oct 2003.

[11] K. Claessen and J. Mårtensson, "An Operational Semantics for Weak PSL," in *Formal Methods in Computer-Aided Design (FMCAD'04)*, A. J. Hu and A. K. Martin, Eds., vol. 3312. LNCS, Springer, Nov. 2004.

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

[13] 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," United States Patent Application 20060136879, June 22nd, 2006.

[14] IEEE Std. 1850-200x Working Group, "Unaddressed Issue #146," *Issues To Be Addressed in IEEE 1850-200x PSL*, 2006.

[15] IEEE Std. 1850-200x Working Group, "Simple Subset Issue #99, Group E.1," *Issues To Be Addressed in IEEE 1850-200x PSL*, 2006.