# Efficient Automata-Based Assertion-Checker Synthesis of PSL Properties

Marc Boulé and Zeljko Zilic
McGill University, Montréal, Québec, Canada
marc.boule@elf.mcgill.ca, zeljko.zilic@mcgill.ca

*Abstract*— **Automata-based methods for generating PSL hardware assertion checkers were primarily considered for use with temporal sequences, as opposed to full-scale properties. We present a technique for automata-based checker generation of PSL *properties* for dynamic verification. A full automata-based approach allows an entire assertion to be represented by a single automaton, hence allowing optimizations which can not be done in a modular approach where sub-circuits are created only for individual operators. For this purpose, automata algorithms are developed for the base cases, and a complete set of rewrite rules is developed and applied for all other operators. We show that the generated checkers are resource-efficient for use in hardware emulation, simulation acceleration and silicon debug.**

## I. Introduction

Assertion-Based Verification (ABV) is emerging as a powerful methodology for design verification [1]. Using temporal logic, a precise description of the expected behavior of a design is modeled, and any deviation from this expected behavior is captured by simulation or by formal methods. Hardware verification assertions are written in verification languages such as PSL (Property Specification Language) or SVA (SystemVerilog Assertions). When used in dynamic verification, a simulator monitors the Device Under Verification (DUV) and reports when assertions are violated. Information on where and when assertions fail is an important aid in the debugging process, and is the fundamental reasoning behind the ABV methodology.

As circuits become more complex, simulation time becomes an important bottleneck in dynamic verification. Simulation acceleration using hardware emulation is increasingly used in the industry – EDA companies such as Cadence and Mentor Graphics offer hardware solutions for high-performance simulation. Hardware emulation involves loading and executing the circuit on reprogrammable hardware, often on an array of programmable logic devices. Once implemented in hardware, the emulator fully exploits the inherent circuit parallelism and the DUV does not have to be processed serially in a conventional simulator.

Assertion languages allow the specification of expressions that do not lend themselves directly to hardware implementations. Such languages allow complex temporal relations between signals to be stated in a compact and elegant form. In order to consolidate assertion-based verification and emulation, a checker generator is used to generate hardware assertion checkers [2], [3]. These checkers are typically expressed in a Hardware Description Language (HDL). An assertion checker is a circuit that captures the behavior of a given assertion, and can be included in the DUV for in-circuit assertion monitoring. A checker generator allows the flexibility of automatically generating custom monitor circuits from any assertion statement.

Assertion languages such as PSL and SVA are built using Boolean expressions, regular expressions for describing sequences, and higher-level properties. In this paper, we focus on properties as used in the PSL language. Since many themes are common to other assertion languages, the presented techniques are not restricted to PSL.

This paper introduces automata techniques and rewrite rules for transforming properties used in assertions into resource-efficient circuits suitable for hardware emulation. These techniques are implemented in our checker generator called MBAC. Ideally, assertion circuits should be compact, fast, and should interfere as little as possible with the DUV, with which it shares the emulator resources. To our knowledge, the only other available stand-alone tool capable of generating hardware checkers from PSL assertions is IBM's FoCs Property Checkers Generator [2], [4]. It will be shown that the circuits generated by MBAC can be significantly more efficient.

The automata produced in [5], [6] can be used to check a property during simulation. These types of checkers indicate the status of the property at the end of simulation/checking only, and are not ideal for debugging purposes. It is much more informative to provide a dynamic trace of the assertion and to signal each assertion failure instance: having a choice of violations to explore eases the debugging process, as some errors may reveal more than others. The time required to generate checkers is larger [5], sometimes significantly so [6] than with FoCs.

A modular approach is employed in [7], [8], whereby sub-modules for each property operator are built and interconnected according to the expression being implemented. Paired interconnects are used, and assertions produce a pair of signals that indicate the status of the assertion. Our circuits produce a single result signal for each assertion and are thus simpler to monitor. We implement liveness properties in a manner similar to FoCs by utilizing a special end-of-execution signal, which can be shared across all assertion circuits. This signal marks the end of time and obligations that were not fulfilled cause the assertion signal to trigger. Since we implement entire assertions as automata, optimizations that can not be seen across module boundaries [3], [7], can be seen in automaton form, and thus help produce minimized checkers.

The contributions of this paper are:

1) Introduction of efficient automata-based (as opposed to the modular approach) methods for compiling a base set of PSL properties, along with an implementation in our checker generator;
2) Introduction of a set of rewrite rules suitable for the remaining non-base properties, thereby simplifying the design of our checker generator (the simple subset of PSL imposes restrictions which do not allow most sugaring definitions [9] to be used as rewrite rules);
3) Presentation of the first published (to our knowledge) implementation of the entire set of PSL properties for dynamic verification, whether automata-based or other.

## II. Background

### A. Assertion Languages and Properties

While there are several modern assertion languages, our tool currently uses PSL (IEEE 1850 Standard, [10]), which is arguably the most complex. We briefly present its features, using the Verilog flavor, with emphasis on PSL properties.

The *Boolean Layer* in PSL is built around the Boolean expressions of the underlying HDL, in addition to symbols *true* and *false*. Let top-level Boolean expressions be represented by single primary symbols labeled $b_i$. Each $b_i$ can be a single signal or a Boolean function of multiple signals. Sequential-Extended Regular Expressions (SEREs) are used to specify temporal chains of events of Boolean primitives. Under a defined clock signal, a simple sequence of Boolean expressions is satisfied in a given clock cycle if in previous clock cycles, each $b_i$ evaluates to true at its respective time point. For example, the SERE $\{b_1; b_2; b_3\}$ evaluates to true (is matched, is observed) if $b_3$ evaluates to true, and in the previous cycle, $b_2$ was true, and before that, $b_1$ was asserted.

**Definition 1**: *SEREs* are defined as follows [9]. If $b$ is a Boolean expression and $r$, $r_1$ and $r_2$ are SEREs, the following expressions are SEREs [1]:

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

The curly brackets are equivalent to parentheses in conventional regular expressions, and the semicolon represents concatenation. In assertion context, concatenation of two Boolean expressions $b_l; b_r$ indicates that the Boolean expression $b_l$ must evaluate to true in one cycle, and $b_r$ must be true in the next cycle. The $[*]$ operator indicates a repetition of zero or more instances, and the $|$ operator corresponds to SERE disjunction.

The colon operator denotes *SERE fusion*, which is a concatenation in which the last Boolean primitive occurring in the first SERE must intersect (both must be true) with the first Boolean primitive occurring in the second SERE. Empty SEREs in either side do not result in a match. The *length matching SERE intersection* (&&) requires that both argument SEREs occur, and that both SEREs start and terminate at the same time. The $[*0]$ operator is the empty SERE, and can be seen as a primitive which spans no clock cycles.

[1]The clocking operator was purposely omitted because PSL expressions will be implicitly clocked to the default clock, specified with PSL's default clock directive.

PSL defines additional syntactic "sugaring" operators which simplify the writing of assertions, but do not add expressive power to the language. The most popular PSL SERE sugaring operators are shown below, which we use as rewrite rules in the checker generator. $b$ is a Boolean expression; $r$ is a SERE; $l$, $h$ and $c$ are nonnegative integers with $h \geq l$; and the $\stackrel{\rightarrow}{=}$ symbol indicates equivalency, with a preferred direction to be used as a rewrite rule. The $^+$ superscript denotes a positive integer.

- $r[+] \stackrel{\rightarrow}{=} r \,;\, r[*]$
- $r[*0] \stackrel{\rightarrow}{=} [*0]$
- $r[*c^+] \stackrel{\rightarrow}{=} r \,;\, r \,;\, \ldots \,;\, r \quad (c \text{ times})$
- $r[*l{:}h] \stackrel{\rightarrow}{=} r[*l] \mid \ldots \mid r[*h]$
- $b[\text{->}] \stackrel{\rightarrow}{=} \{(\sim b)[*] \,;\, b\}$
- $b[\text{->}\, c^+] \stackrel{\rightarrow}{=} \{b[\text{->}]\}[*c]$
- $b[\text{->}\, l^+{:}\, h^+] \stackrel{\rightarrow}{=} \{b[\text{->}]\}[*l{:}h]$
- $b[=c] \stackrel{\rightarrow}{=} \{b[\text{->}\, c]\} \,;\, (\sim b)[*]$
- $b[=l{:}h] \stackrel{\rightarrow}{=} \{b[\text{->}\, l{:}h]\} \,;\, (\sim b)[*]$
- $r_1 \,\&\, r_2 \stackrel{\rightarrow}{=} \{\ \{r_1\} \,\&\&\, \{r_2; [*]\}\ \} \ \mid \ \{\ \{r_1; [*]\} \,\&\&\, \{r_2\}\ \}$

The $[*c]$ and $[*l{:}h]$ operators are known as repetition count and repetition range. The first four operators can be used without the SERE $r$, in which case $r = \textit{true}$ is implied. The $[=]$ operator corresponds to non-consecutive repetition, whereas the $[\text{->}]$ operator is known as goto repetition. The single & is called non-length-matching intersection.

PSL also defines *properties* on sequences and Boolean expressions. When used in properties, SEREs are placed in curly brackets. Sequences are denoted using the symbol $s$, which are formed from SEREs:

$$s := \{r\}$$

SEREs and sequences are different entities, and production rules are more constrained than what was stated in the definition of SEREs (Definition 1). For example, the && operator requires arguments in curly brackets. Since we are mainly concerned with the effect of an operator, the exact syntax rules are deferred to [9].

Properties, like SEREs, are built from a reasonably compact set of operators, to which "sugaring" operators are also added. However, because the simple subset imposes many modifications to the arguments of properties, we will not make the distinction between sugaring and base operators.

Some forms of properties are not suitable for simulation and can only be evaluated by formal methods. The portion of PSL suitable for simulation is referred to as the *simple subset* of PSL. The PSL foundation language properties are shown below (in the Verilog flavor). The properties have been slightly modified to reflect the recent IEEE standard for PSL [10], and are presented with the simple subset modifications (Section 4.4.4 in [10]). Since entire properties are clocked to the same signal edge in our checkers, the clocking property is not shown. Furthermore, PSL's LTL operators $X$, $G$, $F$, $U$ and $W$ are syntactically equivalent to next, always, eventually!, until!

and until, respectively, and are omitted.

**Definition 2**: Let $b$, $b_1$ and $b_2$ be Boolean expressions, let $s$ be a sequence and let $p$, $p_1$ and $p_2$ be properties. If $l$, $h$ and $c$ are nonnegative integers with $h \geq l$, then PSL *foundation language properties* are defined as follows in the simple subset. Here, $^+$ also denotes a positive integer.

- $b$
- $s$
- $p$ abort $b$
- $p_1$ && $p_2$
- $b$ <-> $b$
- $s$ |-> $p$
- $p$ until $b$
- $p$ until! $b$
- $b_1$ before $b_2$
- $b_1$ before! $b_2$
- next $p$
- next! $p$
- next$[c](p)$
- next!$[c](p)$
- next_a$[l{:}h](p)$
- next_a!$[l{:}h](p)$
- next_e$[l{:}h](b)$
- next_e!$[l{:}h](b)$
- always $p$
- eventually! $s$
- $(p)$
- $s!$
- $!b$
- $b$ || $p$
- $b$ -> $p$
- $s$ |=> $p$
- $b_1$ until_ $b_2$
- $b_1$ until!_ $b_2$
- $b_1$ before_ $b_2$
- $b_1$ before!_ $b_2$
- next_event$(b)(p)$
- next_event!$(b)(p)$
- next_event$(b)[c^+](p)$
- next_event!$(b)[c^+](p)$
- next_event_a$(b)[l^+{:}\,h^+](p)$
- next_event_a!$(b)[l^+{:}\,h^+](p)$
- next_event_e$(b_1)[l^+{:}\,h^+](b_2)$
- next_event_e!$(b_1)[l^+{:}\,h^+](b_2)$
- never $s$

Semantics of each property will be discussed in the next section, as it is rewritten or implemented in an automaton form. Properties that appear below the separating line are those for which rewrite rules to the basic cases (above the line) will be devised. Following the above definition of properties, it can be observed [3] that sequences and Boolean expressions can be interpreted in two modes in dynamic verification.

**Definition 3**: *Conditional mode*. Context for which the detection of a sequence or Boolean expression must be performed. For each start condition of a Boolean expression (sequence), the result signal is triggered each and every time the Boolean expression (chain of events described by the sequence) is observed.

**Definition 4**: *Obligation mode*. Context for which the failure of a sequence or Boolean expression must be identified. For each start condition, if the chain of events described by the Boolean expression or sequence does not occur, the result signal is triggered. For a given start condition of a sequence, only the first failure is identified.

For example, in assertions

assert never $\{b_1\,;\,b_2\}$;
assert always $(\{b_1\,;\,b_2\}$ |-> $p_1)$;

both sequences are in *conditional* mode because their presence is used to detect a failing condition. On the other hand, in

assert always $\{b_1\,;\,b_2\}$;
assert always $(b_1$ -> $\{b_2\,;\,b_3\})$;

both sequences are in *obligation* mode because their failure to occur is used to trigger a condition.

As shown in Definition 2, Boolean expressions and sequences can be used directly as properties. When such a case occurs, a start condition indicates that the Boolean expression or sequence should occur, and that a non-occurrence constitutes a violation of the expected behavior. The obligation that is placed on sequences is not strong, meaning that if the execution terminates and the sequence did not complete, no error is signaled. A strong sequence, $s!$, is used to indicate that each start condition should see the sequence complete. For these types of properties to be evaluated properly, an end-of-execution (EOE) signal must be provided by the user. This signal is normally at logic-0 and must be asserted for at least one clock cycle when the simulation / emulation has completed. If an EOE occurs during the processing of a strong sequence, the assertion signal will trigger.

As an example of the use of properties in concrete hardware verification, consider the property encountered in the verification of bus arbiters:

always $(\{\text{requestA}\}$ |-> $\{(\sim\text{grantA})[*0{:}15]\ ;\ \text{grantA}\})$

This property states that when a request is issued to the arbiter, agent A will receive a bus grant within 16 clock cycles. If the stated condition is not satisfied, an assertion error occurs.

A *checker generator* is the tool that transforms assertions into assertion monitor-circuits (checkers), directly usable in hardware emulation. Individual assertions, once converted to circuit form, are also referred to as *assertion circuits*.

## *B. Automata for Sequences and Boolean Expressions*

An automaton can be depicted by a directed graph, where vertices are states, and the conditions for transitions among the states are inscribed on edges [11]. In our case, the transition conditions are symbols which represent complete Boolean-layer expressions. For a given assignment of various symbols, all conditions that are true will cause a transition into a new *set* of active states, thereby producing a non-deterministic automaton. A sequence can be converted to an equivalent finite automaton in a recursive manner [12]. First, terminal automata are built for the Boolean expressions. Next, these automata are recursively combined according to the operators used in a sequence.

In the automaton that represents an assertion, an assertion violation is reported each time a final state is activated. To provide more useful debug information, our automata algorithms are designed such that the assertion result signal does not simply indicate a yes/no answer obtained at the end of execution, but rather a continuous and dynamic report of when the assertion has failed.

Henceforth, $\mathcal{A}\big(expr\big)$ and $\mathcal{A}_C\big(expr\big)$ indicate the construction of an automaton for a given PSL expression. We will use symbols $s$ and $b$ for representing sequences and Boolean expressions respectively. Subsequently, $\mathcal{A}\big(s\big)$ and $\mathcal{A}\big(b\big)$ denote *obligation mode* automata for sequences and Boolean expressions, respectively, in accordance with Definition 4. This mode will be employed when Boolean expressions or sequences

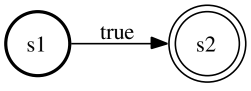

Fig. 1. $\mathcal{A}_C\big(\textit{true}\big)$, $\mathcal{A}\big(\textit{false}\big)$

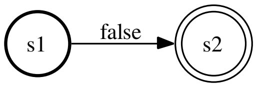

Fig. 2. $\mathcal{A}_C\big(\textit{false}\big)$, $\mathcal{A}\big(\textit{true}\big)$

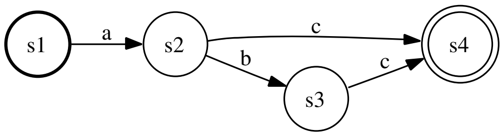

Fig. 3. $\mathcal{A}_C\big(\{a\,;b[*0:1]\,;c\}\big)$

are used in properties. Some properties such as never and suffix implication also require *conditional mode* sequences and Boolean expressions, for which the corresponding automata are denoted $\mathcal{A}_C\big(s\big)$ and $\mathcal{A}_C\big(b\big)$. Since a Boolean expression $b$ can be seen as the sequence $\{b\}$, the construction of automata for Boolean expressions in both modes is subsumed by the construction of automata for sequences. Further details of constructing automata for sequences appear in [12], but are not required for this paper.

Figures 1 and 2 show simple automata for the Boolean expressions *true* and *false*, in both modes. In Figure 2, since the *false* symbol can never be true, the automaton never reaches the final state. Figure 3 shows a conditional mode automaton for detecting the sequence $\{a\,;b[*0:1]\,;c\}$. Figure 4 shows how the same sequence is processed in obligation mode by an automaton. When a conditional mode automaton reaches a final state (double circle), the expression represented by the automaton has been *detected*. When an obligation mode automaton reaches a final state, the *first failure* of the expression has been caught. The state in a bold circle is the start state of the automaton.

## III. Transforming Properties into Circuits

We now show how to transform properties into automata, for subsequent conversion to circuit form. The resulting HDL circuit descriptions become the checkers that are responsible for monitoring the behavior that is modeled by the assertions. Implementing an automaton in hardware is done in two parts. First, each state signal is sampled by a flip-flop (FF). The FF's output is referred to as the sampled state-signal. Second, a state signal is defined as a disjunction of the edge signals that hit a given state. An edge signal is a conjunction of the edge's symbol with the sampled state signal from which the edge

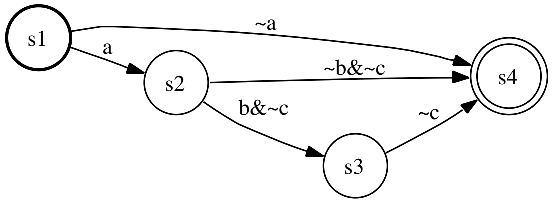

Fig. 4. $\mathcal{A}\big(\{a\,;b[*0:1]\,;c\}\big)$

originates. The signal which is returned by the automaton, called result signal, is a disjunction of the *state signals* of the final states (as opposed to the sampled state signals). This is an important distinction because when the automaton represents the left side of a non-overlapping temporal implication, for example, this result signal is used as the start condition for the right side, which must start immediately when the start condition is detected. In sum, automata are implemented in hardware using combinational logic and flip-flops. In this section, target properties from Definition 2 will be underlined.

To show how properties and automata relate, compiling property $\underline{(p)}$ simply involves the automaton returned by the argument of this property, namely $\mathcal{A}\big(p\big)$. Since the parentheses are used for grouping:

$$\mathcal{A}\big((p)\big) = \mathcal{A}\big(p\big)$$

Compilation of a PSL property involves recursively scanning the syntax tree of the PSL expression. Each node returns an automaton describing the behavior of the sub-property rooted at that node. The parent then builds its own sub-property automaton from its children automaton(s), using a variety of transformations and operations.

Properties are inherently in obligation mode; however, Definition 4 is too strict for use in properties. To provide more debugging information for properties, the obligation is not limited to the first failure for each start condition. For example, the property never $\{a\}$ is made to trigger every time $a$ is observed. Since properties are meant to catch failures, the PSL directive assert $p$ does not affect the automaton for $p$:

$$\mathcal{A}\big(\,\text{assert}\ p\,\big) = \mathcal{A}\big(p\big)$$

An assertion signal is normally at logic-0, and triggers when a violation is observed.

In general, automata equivalency $\equiv$ means that both automata detect the same patterns, and does not imply that both automata are identical. In automata theory, equivalent automata are said to accept the same language. The = sign above emphasizes that the automata are in fact identical.

### A. Implementation of Base Cases

The approach for implementing the base cases consists in taking the automaton(s) returned by the arguments of a property, and then building a single resulting automaton for the property and its arguments. This way, an entire assertion can be represented by a single finite automaton.

The properties $\underline{!b}$ and $\underline{b_1 <\!\!-\!\!> b_2}$ are in effect relegated to the Boolean layer in the simple subset. The negation and equivalency of properties – as allowed in full PSL – create properties that are not suitable for monotonically advancing time. The simple subset version of these properties are therefore handled in the $\mathcal{A}\big(b\big)$ construction. Equivalency is rewritten as the Boolean expression: $(\sim b_1 \mid b_2)$ & $(\sim b_2 \mid b_1)$.

The automata for properties $\underline{b}$ and $\underline{s}$ are obtained by constructing $\mathcal{A}\big(b\big)$ and $\mathcal{A}\big(s\big)$ respectively. Implementing property $\underline{s!}$ involves constructing $\mathcal{A}\big(s\big)$ with a slight modification. The typical obligation mode automaton for $s$ is constructed, with added edges that cause the automaton to transition from any active state to a final state upon activation of the end-of-execution signal. If the automaton is processing a sequence when the EOE occurs, a completion is not fulfilled and an error is detected.

Handling the $\underline{p \text{ abort } b}$ property involves modifying the automaton of $p$. When the abort operator is encountered in the syntax tree, $\mathcal{A}(p)$ is built and a new primary symbol for $\sim b$ is created. This primary symbol is then and-ed to each symbol on the edges in $\mathcal{A}(p)$. When an abort is asserted, all transitions in the automaton are disabled and the automaton is in effect reset.

Constructing an automaton for the $\underline{p_1 \ \&\& \ p_2}$ property involves creating an automaton for:

$$\mathcal{A}(p_1) \mid \mathcal{A}(p_2)$$

The $\mid$ operator represents automata disjunction, a well-known operator [11]. The disjunction is required because a failure by either sub-property constitutes a failure for the && property. At this point, we start noticing a form of negation between conditional behavior and obligation behavior. This was seen earlier, when the example $\mathcal{A}_C(\textit{true})$=$\mathcal{A}(\textit{false})$ was shown. The negation behavior was also observed in [5]. In our work, the negation is not perfect given the dynamic behavior we wish our circuits to offer. In dynamic property checking, a proper negation between both modes is not apropriate because a sequence that failed once could continue to fail for the remainder of execution, which is inadequate for debugging purposes.

Although rewriting $\underline{p \text{ until } b}$ as $\{(\sim b)[+] : \{p\}\}$ might appear logical, this can not be included as a rewrite rule in the next subsection. Fusion with a property is not valid in PSL syntax. We keep the equivalency here however, because it illustrates what is done in the kernel. The algorithm for the fusion of two automata was developed in the context of SEREs [12]; however, the algorithm can be used on property automata as well, given that sequence automata and property automata are based on the same automata techniques. In other words, the automata fusion algorithm can be seen as a general automata operator, in the same class as automata disjunction, which was used above for the && property. For the until rewrite, all that must be done is to ensure that the semantics of the fusion operator is appropriate and provides the desired behavior in the context of properties, which we explain next (full proofs are omitted for reasons of space).

The until operator states that property $p$ must be true on each cycle, up-to, but not including, $b$ being true. The overlap created by the fusion has the effect of sending a start condition to $p$ for each cycle of consecutive $\sim b$'s. In our dynamic semantics for the until operator, the property is allowed to fail multiple times for a given start condition when $b$ is continuously false.

Suffix implication is often used in assertions, and will be utilized in some of the rewrite rules of the next subsection. Implementing the $\underline{s \mid\!-\!> p}$ suffix implication can be summarized by the following strategy:

$$\mathcal{A}_C(s) : \mathcal{A}(p)$$

When used in the context of automata, the ":" symbol denotes the actual automata fusion algorithm, the same algorithm that is invoked when the SERE fusion operator ":" is encountered. Using fusion in properties does not create unwanted side-effects, considering that the empty SERE can not cause a match on either side of fusion. In properties, a conditional or obligation mode automaton's start state can never be a final state; this creates automata behavior that is consistent with the formal semantics of PSL in Appendix B in [9]. As an example, when a sequence automaton's start state is a final state, and this sequence is used as a precondition to suffix implication, the empty match can not cause the post-condition to be enforced. When a conditional mode sequence automaton is used at the property level and its start state is a final state, the state is made non-final. When an obligation mode sequence automaton is built for a sequence that can not hold, the automaton from Figure 1 is returned to the parent operator. This allows the proper processing of the following assertions:

assert always $\{s\} \mid\!-\!> \{\ \{b\}\ \&\&\ \{b;b\}\ \}$;
assert always $\{b[*0{:}1]\} \mid\!-\!> p$;

In the first line, the length-matching intersection results in a post-condition automaton identical to Figure 1. The assertion thus fails whenever $s$ is observed. In the second line, $b$ must be asserted *once* in order for the post-condition to be enforced. The null intersection and empty repetition ($[*0]$) are known as *degenerate* sequences. Fusion is also used in the algorithm for suffix implication presented in [5].

Property replication using the forall operator[2] is implemented by looping through each value of the forall value set, and performing the automata disjunction of each sub-property's automaton.

In our work, a full automaton approach allows the production of efficient automata. Consider the following examples, which yield *identical* automata, as opposed to the modular approach which will not simplify the goto repetition, given the preceeding always operator. A modular approach will generate more checker code in the second example than in the first, even though both assertions are semantically identical.

assert always $\{a\} \mid\!=\!> b$;
assert always $\{a[-\!>]\} \mid\!=\!> b$;

## B. Rewrite Rules for Properties

Most properties from Definition 2 do not need to be explicitly handled in the checker generator kernel. When such properties can be expressed using the base cases from the previous subsection, they are dynamically rewritten when encountered during checker generation. The rules by which these properties are rewritten are called *rewrite rules*. In some cases, the semantics of the property can be captured by a single sequence, for others, sequences and base properties are used.

As indicated in the introduction, using the sugaring definitions from Appendix B in [9] as rewrite rules is generally

[2]The forall operator is not a *foundation language property* as such [9].

not feasible because of the restrictions imposed by the simple subset. For this purpose, we introduce a set of rewrite rules that is suitable for the simple subset of PSL, within the context of dynamic verification. The rules are not intended to extend upward to full PSL. Although a few rewrite rules may appear intuitive, they are nonetheless included for completeness. The following sugaring definition shows an example of why such definitions can generally not be used as rewrite rules:

$$\textsf{always } p = \neg \textsf{ eventually! } \neg p \qquad (\mathrm{G}\, p = \neg\, \mathrm{F} \neg p \;\; [9])$$

The above sugaring definition for always can not be used in the simple subset because negating a property is not permitted. In this section, rewrite rules that are compatible with the simple subset are developed. In some cases, the easiest way to handle an operator is by rewriting it using a more complex operator. Since the more complex operator has to be handled, we avoid writing particular code for the simpler cases. For example, rewriting next_a using next_event_a may appear overly complex; however, since next_event_a already exists and must be supported, it subsumes all simpler forms of this family of operators.

Shown below are the rewrite rules that we have developed for the simple subset of PSL. A brief explanation follows each rule, while full proofs are omitted for space reasons.

$$\underline{b \;||\; p} \quad \overset{\rightarrow}{=} \quad (\sim b) \;\text{->}\; p$$

In the simple subset, one of the properties used in disjunction must be a Boolean expression. For simplicity, in the rule above and in Definition 2, the Boolean expression is shown as the left argument. This rewrite rule is based on the fact that if the Boolean expression is not true, then the property must be true; else the property is automatically true because $b$ is true.

$$\underline{b \;\text{->}\; p} \quad \overset{\rightarrow}{=} \quad \{b\} \;|\text{->}\; p$$

Since a Boolean expression can be easily expressed as a sequence, we can rewrite the above form of implication to a suffix implication.

$$\begin{array}{rcl} \underline{\textsf{always } p} & \overset{\rightarrow}{=} & \{[+]\} \;|\text{->}\; p \\ \underline{\textsf{never } s} & \overset{\rightarrow}{=} & \{[+] : s\} \;|\text{->}\; \mathit{false} \end{array}$$

As explained in the previous subsection, suffix implication has a conditional-mode sequence as a precondition (prefix), and a property as a post-condition (suffix). When a property must always be true, it can be seen as the post-condition of a suffix implication with a constantly asserted precondition ([+] is sugaring for *true*[+]). When a sequence must not occur, a property which fails instantly is triggered upon the sequence detection. Because suffix implication does not have a clock cycle delay between pre and post-conditions, these rewrites offer the correct timing.

$$\begin{array}{rcl} \underline{\textsf{next } p} & \overset{\rightarrow}{=} & \textsf{next}[1](p) \\ \underline{\textsf{next! } p} & \overset{\rightarrow}{=} & \textsf{next!}[1](p) \end{array}$$

The above rewrites use a slightly more explicit form of next operators. Since the right-hand side of this rule is not terminal, it is subsequently rewritten using another rule. This recursive process continues until no more rewrites apply, and we reach sequences, Boolean expressions or base cases from the previous subsection.

$$\underline{\textsf{eventually! } s} \quad \overset{\rightarrow}{=} \quad \{[+] : s\}!$$

Rewriting the eventually! operator is done by enforcing that the sequence $s$ must complete before the end of execution. The sequence may start at any time after it is triggered, hence the fusion with [+]. This rewrite does not apply to degenerate sequences, which are handled separately.

$$\underline{p \textsf{ until! } b} \quad \overset{\rightarrow}{=} \quad (p \textsf{ until } b) \;\&\&\; (\{b[\text{->}]\}!)$$

Property conjunction and the weak version of the until property are base cases. The strong version of this property is created by using the weak version, and 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 $b$. If the end-of-execution occurs before the releasing condition has manifested itself, the assertion will trigger, even though the weak until may have always held.

$$\begin{array}{rcl} \underline{b_1 \textsf{ until\_ } b_2} & \overset{\rightarrow}{=} & \{(b_1)[+] : (b_2)\} \\ \underline{b_1 \textsf{ until!\_ } b_2} & \overset{\rightarrow}{=} & \{(b_1)[+] : (b_2)\}! \end{array}$$

The behavior of the overlapped until_ properties can be captured by sequences, given that no properties are allowed in their arguments in the simple subset. It should be noted that the rewrites above create sequences in the place of the operators they replace, and are inherently in obligation mode. The rewrite rules state that $b_2$ must occur fused with a block of any number of consecutive $b_1$'s. Since fusion is used, $b_1$ must be true for at least one cycle concurrently with $b_2$, therefore the overlap required by the "_" is also well modeled.

$$\begin{array}{rcl} \underline{b_1 \textsf{ before } b_2} & \overset{\rightarrow}{=} & \{(\sim b_1 \& \sim b_2)[*] \;;\; (b_1 \& \sim b_2)\} \\ \underline{b_1 \textsf{ before! } b_2} & \overset{\rightarrow}{=} & \{(\sim b_1 \& \sim b_2)[*] \;;\; (b_1 \& \sim b_2)\}! \\ \underline{b_1 \textsf{ before\_ } b_2} & \overset{\rightarrow}{=} & \{(\sim b_1 \& \sim b_2)[*] \;;\; b_1\} \\ \underline{b_1 \textsf{ before!\_ } b_2} & \overset{\rightarrow}{=} & \{(\sim b_1 \& \sim b_2)[*] \;;\; b_1\}! \end{array}$$

The before family of properties can also be modeled by obligation mode sequences. The overlapped versions state that $b_1$ must be asserted before or simultaneously with $b_2$. The strong versions also expect $b_2$ to occur before EOE, if not the assertion will trigger.

$$\begin{array}{rcl} \underline{\textsf{next}[c](p)} & \overset{\rightarrow}{=} & \textsf{next\_event}(\mathit{true})[c+1](p) \\ \underline{\textsf{next!}[c](p)} & \overset{\rightarrow}{=} & \textsf{next\_event!}(\mathit{true})[c+1](p) \end{array}$$

Alternate rewrite rules for next$[c](p)$ could also be developed using next_a. The above form is chosen here so that the "$c+1$" artifact can be better explained; this explanation will also apply to some of the rewrite rules that will follow. 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. Therefore, when utilizing next_event(*true*) to subsume next, an extra cycle must be added, hence the increment by one on $c$. Another reasoning shows the consistency between the operators: we observe that next[0]$(p)$ could not be modeled without the increment because next_event$(b)[c^+](p)$ requires a positive count. Incidentally, next[0]$(p)$ is equivalent to $(p)$.

$$\begin{array}{rcl} \underline{\mathsf{next\_a}[l{:}h](p)} & \stackrel{\rightarrow}{=} & \mathsf{next\_event\_a}(\mathit{true})[l{+}1:h{+}1](p) \\ \underline{\mathsf{next\_a!}[l{:}h](p)} & \stackrel{\rightarrow}{=} & \mathsf{next\_event\_a!}(\mathit{true})[l{+}1:h{+}1](p) \\ \underline{\mathsf{next\_e}[l{:}h](b)} & \stackrel{\rightarrow}{=} & \mathsf{next\_event\_e}(\mathit{true})[l{+}1:h{+}1](b) \\ \underline{\mathsf{next\_e!}[l{:}h](b)} & \stackrel{\rightarrow}{=} & \mathsf{next\_event\_e!}(\mathit{true})[l{+}1:h{+}1](b) \end{array}$$

The above family of rewrite rules for the next_a and next_e properties are based on the fact that the next_event version is a more general case, with the "+1" adjustment to handle the mapping of the Boolean *true*, as indicated previously.

$$\begin{array}{rcl} \underline{\mathsf{next\_event}(b)(p)} & \stackrel{\rightarrow}{=} & \mathsf{next\_event}(b)[1](p) \\ \underline{\mathsf{next\_event!}(b)(p)} & \stackrel{\rightarrow}{=} & \mathsf{next\_event!}(b)[1](p) \end{array}$$

When no count is specified for the above form of next_event, a count of 1 is implicit.

$$\begin{array}{rcl} \underline{\mathsf{next\_event}(b)[c^+](p)} & \stackrel{\rightarrow}{=} & \mathsf{next\_event\_a}(b)[c{:}c](p) \\ \underline{\mathsf{next\_event!}(b)[c^+](p)} & \stackrel{\rightarrow}{=} & \mathsf{next\_event\_a!}(b)[c{:}c](p) \end{array}$$

The strategy behind the above rewrites is to utilize the next_event_a form, with identical upper and lower bounds for the range. The semantics of next_event_a is explained next.

$$\underline{\mathsf{next\_event\_a}(b)[l^+{:}h^+](p)} \quad \stackrel{\rightarrow}{=} \quad \{b[{-}{>}l{:}h]\} \mid{-}{>} (p)$$

The next_event_a property 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 a precondition to the property. This in effect sends a start condition to the property each time $b$ occurs within the specified range after the current property received its start condition. These types of properties are notorious for creating highly pipelined and temporally complex chains of events. The automaton approach is well suited to handle these situations.

$$\underline{\mathsf{next\_event\_a!}(b)[l^+{:}h^+](p)} \quad \stackrel{\rightarrow}{=} \quad \mathsf{next\_event\_a}(b)[l{:}h](p) \ \&\& \ \{b[{-}{>}h]\}!$$

This is 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 $h$ bound indicates that for each start condition of the next_event_a, all $h$ occurrences of the $b$ event must occur before execution terminates.

$$\begin{array}{rcl} \underline{\mathsf{next\_event\_e}(b_1)[l^+{:}h^+](b_2)} & \stackrel{\rightarrow}{=} & \{b_1[{-}{>}l{:}h] : b_2\} \\ \underline{\mathsf{next\_event\_e!}(b_1)[l^+{:}h^+](b_2)} & \stackrel{\rightarrow}{=} & \{b_1[{-}{>}l{:}h] : b_2\}! \end{array}$$

The next_event_e properties 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 obligation mode sequence has completed and will not indicate a failure. The strong version is created by using a strong sequence.

$$\underline{s \mid{=}{>} p} \quad \stackrel{\rightarrow}{=} \quad \{s \,;\, \mathit{true}\} \mid{-}{>} p$$

This rewrite rule follows from the sugaring definition in Appendix B in [9]. The simple subset does not affect this definition, therefore it can be used directly as a rewrite rule.

## IV. Experimental Results

In this section, the circuits produced by the MBAC checker generator are evaluated using various test assertions. The hardware comparison metrics involve synthesizing the assertion circuits using ISE 6.2.03i from Xilinx, for a XC2V1500–6 FPGA. The number of flip-flops (FF) and four-input lookup tables (LUT) required by a circuit is of primary interest when assertion circuits are to be used in hardware. The maximum operating frequency (MHz) for the worst clk-to-clk path is also reported. MBAC's final result signals are sampled by a FF, and thus have the same timing as FoCs' circuits.

The assertions used for evaluating checker generators typically do not contain complex Boolean expressions because such expressions have no effect on the temporal complexity of assertions. Without loss of generality, the Boolean layer is abstracted away using simple signal names $a$, $b$, etc. Furthermore, temporally simple assertions such as those used for verifying bus protocols (e.g., AMBA bus assertion from from [13]) are not informative for evaluating a checker generator, as they span very few clock cycles.

The FoCs and MBAC checker generators are evaluated with the set of assertions shown in Table I. The synthesis results are reported in Table II, where N.S.Y. indicates "Not Supported Yet". With the exception of P14 and P19, in the test cases where FoCs succeeds, both tools produce functionally equivalent circuits. This was verified by exercising the checker circuits using a testbench which produces $10^5$ pseudo-random test vectors. For each clock cycle of the simulation, the circuits produced by both tools offer the same behavior on each clock cycle, for a given assertion. Signal probabilities were adjusted in order for the assertions to trigger reasonably often. This method is not a proof that the circuits are functionally equivalent, however it does offer reasonable assurance.

For the P14 test case, slight differences in behavior were noticed due to the semantics of the until operator. When handling $p$ until $b$, it is up to the tool's architect to decide whether to flag all failures of $p$ before $b$ occurs, or to flag only the first one. This flexibility is expected in dynamic verification with PSL, and may occur with other operators as well. With P19, we believe our circuit is operating properly. The results show that in all cases, our circuits are more resource-efficient than FoCs'.

## V. Conclusion and Continuing Work

We have presented methods suitable for efficiently implementing PSL properties in checker generators. The base cases were handled using automata techniques which build on our work using automata for SEREs. A set of rewrite rules that account for all peculiarities of PSL were also devised. These rewrites represent the simplest way to support such operators in the kernel of the checker generator. The rewrite rules presented are independent of the automata techniques

TABLE I
BENCHMARKING PROPERTIES.

| **Property** (note: properties P1 to P6 are from [8]) | |
|---|---|
| P1: always (a –> next (next_a[2:10](next_event(b)[10]((next_e[1:5](d)) until (c))))) | |
| P2: always ((a –> next(next[10](next_event(b)((next_e[1:5](d)) until (c))))) \|\| e) | |
| P3: (always (a –> next(next[10](next_event(b)((next_e[1:5](d)) until (c)))))) && (always (e –> (next_event_a(f)[1:4](next((g before h) until (i)))))) | |
| P4: always (a –> (next_event_a(b)[1:4](next((d before e) until (c))))) | |
| P5: always (a –> (next_event(c)((next_event_e(d)[2:5](e)) until (b)))) | |
| P6: always (a –> next_event_e(b)[1:6](c)) | P7: always {a;b;c} \|=> never {d[*0:3];e} |
| P8: always a –> eventually! b | P9: always (a –> {[*0:7];b}) abort ∼c |
| P10: always a –> next_a![2:4](b) | P11: always a –> next_e![2:4](b) |
| P12: always a –> next_event_e!(b)[2:4](c) | P13: always a –> next_event_a!(b)[5:10](c) |
| P14: always a –> (b until! c) | P15: always a –> ({b;c} until! d) |
| P16: always ({a;b} \|–> eventually! {c;d}) abort e | P17: never {a;[*];b;c[+]} |
| P18: assert always (a –> ( (eventually! b[*5]) abort c) ) abort d | P19: always a –> (b before!_ c) |
| P20: always ( (a –> (b before c)) && (c –> eventually! {b;d}) ) abort e | P21: always ( ({a;d} \|–> next_e[2:4](b)) until c ) |
| P22: always {a;b[*0:2];c} \|=> ({d[*2]} \|–> next ∼e) | P23: always (e \|\| (a –> ({b;c} until d))) |

TABLE II
HARDWARE METRICS FOR PROPERTY CHECKERS.

| **Property** | **MBAC** | | | **FoCs** | | |
|---|---|---|---|---|---|---|
| “assert P$x$;” | FF | LUT | MHz | FF | LUT | MHz |
| P1 | **26** | **11** | **428** | 290 | 196 | 378 |
| P2 | **17** | **8** | 622 | 24 | 15 | 622 |
| P3 | **26** | **22** | **355** | 53 | 44 | 357 |
| P4 | **10** | **17** | **425** | 30 | 31 | 357 |
| P5 | **8** | **12** | 404 | 48 | 49 | **454** |
| P6 | **7** | **9** | 487 | 12 | 13 | **622** |
| P7 | **7** | **8** | **487** | N.S.Y. | | |
| P8 | 2 | 3 | 622 | 2 | 3 | 622 |
| P9 | **8** | **9** | **680** | N.S.Y. | | |
| P10 | **5** | **4** | **487** | N.S.Y. | | |
| P11 | **5** | **6** | **487** | N.S.Y. | | |
| P12 | **5** | **9** | **425** | N.S.Y. | | |
| P13 | **11** | **11** | **378** | N.S.Y. | | |
| P14 | **2** | **4** | **622** | 3 | 5 | 487 |
| P15 | **3** | **5** | **483** | N.S.Y. | | |
| P16 | **4** | **6** | **428** | N.S.Y. | | |
| P17 | **3** | **3** | **622** | 9 | 8 | 428 |
| P18 | **6** | **13** | **428** | N.S.Y. | | |
| P19 | **2** | **4** | **622** | 3 | 5 | 487 |
| P20 | **4** | **8** | **428** | N.S.Y. | | |
| P21 | **6** | **6** | **680** | N.S.Y. | | |
| P22 | **7** | **7** | **622** | N.S.Y. | | |
| P23 | **3** | **5** | **487** | N.S.Y. | | |

employed, and can be implemented in any tool that utilizes PSL for dynamic verification.

We are currently exploring a more efficient implementation for eventually!, which involves multiple automata and various non-standard modifications. In our approach, conditional mode automata remain nondeterministic and are thus more resource-efficient when implemented in circuit form. By implementing eventually! with a conditional mode automaton, as opposed to the obligation mode automaton implied by our rewrite rule, we have observed that a more efficient implementation is possible in most test cases.

Future work involves using theorem provers such as HOL or PVS to formally prove the rewrite rules in section III-B. The equality of rewrites could be modeled as conjectures using the PSL embedding in HOL [6]; however, proving the conjectures is a non-obvious semi-automated procedure. We are also updating our checker generator to consolidate a full automaton approach with the various debug enhancements introduced in [14].

## REFERENCES

[1] H. Foster, A. Krolnik, and D. Lacey, *Assertion-Based Design, $2^{nd}$ ed.* Kluwer Academic Publishers, 2004.

[2] Y. Abarbanel, I. Beer, L. Glushovsky, S. Keidar, and Y. Wolfsthal, “FoCs: Automatic Generation of Simulation Checkers from Formal Specifications,” *Conference on Computer Aided Verification*, pp. 538–542, 2000.

[3] M. Boulé and Z. Zilic, “Incorporating Efficient Assertion Checkers into Hardware Emulation,” *IEEE International Conference on Computer Design (ICCD–2005)*, pp. 221–228, 2005.

[4] IBM AlphaWorks, “FoCs Property Checkers Generator ver. 2.03,” *www.alphaworks.ibm.com/tech/FoCs*, 2006.

[5] S. Gheorghita and R. Grigore, “Constructing Checkers from PSL Properties,” *15th International Conference on Control Systems and Computer Science (CSCS15)*, vol. 2, pp. 757–762, 2005.

[6] M. Gordon, J. Hurd, and K. Slind, “Executing the Formal Semantics of the Accellera Property Specification Language by Mechanised Theorem Proving,” *Lecture Notes in Computer Science*, vol. 2860, pp. 200–215, Oct. 2003.

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

[8] D. Borrione, M. Liu, K. Morin-Allory, P. Ostier, and L. Fesquet, “On-Line Assertion-Based Verification with Proven Correct Monitors,” *ITI 3rd International Conference on Information & Communications Technology (ICICT 2005)*, pp. 123–143, 2005.

[9] Accellera, “Property Specification Language Reference Manual, v.1.1,” *www.eda.org/vfv/docs/PSL-v1.1.pdf*, 2004.

[10] IEEE Computer Society, “IEEE Standard for Property Specification Language (PSL),” *IEEE Standards*, 2005.

[11] J. Hopcroft, R. Motwani, and J. Ullman, *Introduction to Automata Theory, Languages and Computation, $2^{nd}$ ed.* Addison–Wesley, 2000.

[12] M. Boulé and Z. Zilic, “Efficient Automata–Based Assertion–Checker Synthesis of SEREs for Hardware Emulation,” *(under review)*, 2006.

[13] B. Cohen, S. Venkataramanan, and A. Kumari, *Using PSL/ Sugar for Formal and Dynamic Verification.* Los Angeles, California: VhdlCohen Publishing, 2004.

[14] M. Boulé, J. Chenard, and Z. Zilic, “Adding Debug Enhancements to Assertion Checkers for Hardware Emulation and Silicon Debug,” *(to appear) IEEE International Conference on Computer Design*, 2006.