Automata-Based Assertion-Checker Synthesis of PSL Properties

MARC BOULÉ and ZELJKO ZILIC
McGill University

Assertion-based verification with languages such as PSL is gaining in importance. From assertions, one can generate hardware assertion checkers for use in emulation, simulation acceleration and silicon debug. We present techniques for checker generation of the complete set of PSL *properties*, including all variants of operators, both strong and weak. A full automata-based approach allows an entire assertion to be represented by a single automaton, hence allowing optimizations that can not be done in a modular approach where subcircuits 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 derived for other operators. Automata splitting is introduced for an efficient implementation of the eventually! operator.

Categories and Subject Descriptors: B.5.2 [**Register-Transfer-Level Implementation**]: Design Aids—*Verification*; J.6 [**Computer-Aided Engineering**]: Computer-Aided Design (CAD); F.4.3 [**Mathematical Logic and Formal Languages**]: Formal Languages—*Classes defined by grammars or automata*

General Terms: Verification, Algorithms

Additional Key Words and Phrases: PSL, Assertion-Based Verification, automata, hardware, emulation, assertion checkers

ACM Reference Format:

Boulé, M. and Zilic, Z. 2008. Automata-based assertion-checker synthesis of PSL properties. ACM Trans. Des. Autom. Electron. Syst. 13, 1, Article 4 (January 2008), 21 pages. DOI = 10.1145/1297666.1297670 http://doi.acm.org/10.1145/1297666.1297670

1. INTRODUCTION

Assertion-Based Verification (ABV) is a powerful methodology for design verification [Foster et al. 2004]. Using temporal logic, a precise description of the expected behavior of a design is captured, and any deviation from this expected behavior is verified by simulations or by formal methods. Hardware assertions are written in verification languages such as PSL (Property Specification Language) or SVA (SystemVerilog Assertions). When used in dynamic

Author's address: M. Boulé, McGill University, McConnell Building, Room 633, 3480 University Street, Montréal, Québec, H3A 2K6, Canada, email: marc.boule@elf.mcgill.ca.

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 a bottleneck in dynamic verification. Simulation acceleration and hardware emulation is increasingly used in the industry in the form of hardware products for high-performance emulation. 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 [Abarbanel et al. 2000; Boulé and Zilic 2006]. These checkers are typically expressed in a Hardware Description Language (HDL). An assertion *checker* (or assertion circuit) 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 can be seen as a synthesizer of monitor circuits from assertions, for use in verification, silicon debug and online monitoring.

This paper introduces automata techniques and rewrite rules for transforming PSL *properties* used in assertions into efficient checker circuits. These techniques are implemented in our checker generator called MBAC. Assertion circuits should be compact, fast and should interfere as little as possible with the DUV, with which they share the resources. There are two other similar stand-alone tools in the literature for generating hardware checkers from PSL assertions. IBM's FoCs Property Checkers Generator [Abarbanel et al. 2000; IBM AlphaWorks 2006] (v.2.04) is the oldest such tool. Concurrently to us, a checker generator is being developed at the TIMA laboratory, with results comparable to FoCs 2.02: sometimes better, sometimes worse, and in the best cases, approximately four times better for large assertions (Borrione et al. [2005] and elsewhere).

The automata produced in Gheorghita and Grigore [2005] and Gordon et al. [2003] 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. In dynamic verification, it is much more informative to provide a dynamic trace of the assertion and to signal each assertion failure instance.

A modular approach was employed in a previous version of our tool, and also in Borrione et al. [2005]; Das et al. [2006]; and Morin-Allory and Borrione [2006a], whereby submodules for each property operator are built and interconnected according to the expression being implemented. Since we implement entire assertions as automata, in our case the optimizations can be applied across module boundaries, and thus help produce minimized checkers. As in FoCs, we also utilize an end-of-execution signal that marks the end of time for strong properties. This is required so that any unfulfilled obligations can trigger the checker to indicate an error.

The rewrite rules and automata-based checker synthesis for base cases introduced in Boulé and Zilic [2006] are presented here in a more comprehensive and succinct manner. Major modifications to the until family of operators are integrated herein: future PSL modifications are planned to relax the limitations to the until operators, which we now support; and the until operator is now presented as a rewrite rule, which simplifies its explanations. Another improvement concerns the eventually! operator, which is no longer implemented using a rewrite rule, but rather with a new automata splitting operation. Experimental results show the advantages of this new strategy, and in general, it will be shown that the circuits generated by MBAC can be significantly more efficient, and can support all operators.

2. BACKGROUND

The two most popular assertion languages are SystemVerilog Assertions (SVA) and PSL, now standardized as IEEE Std. 1850 [2005]. Below we briefly present PSL in Verilog flavor. Many of the themes presented in this paper apply to SVA assertions as well. The clocking operator is purposely omitted, as assertions will be clocked using the default clock directive.

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.

Definition 2.1. *SEREs* are defined as follows. If b is a Boolean expression and r, r_1 and r_2 are SEREs, the following expressions are *SEREs*:

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

Some aspects of SERE notation are equivalent to conventional regular expressions (REs): the $[*]$ operator is a repetition of zero or more instances, and the $|$ operator corresponds to SERE disjunction. The $[*0]$ operator denotes a primitive that spans no clock cycles and is referred to as the empty SERE; it is similar to the ϵ symbol in REs. Further, the curly brackets are equivalent to parentheses in REs. 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 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. The *length matching SERE intersection* (&&) requires that both argument SEREs occur, and that both SEREs start and terminate at the same time.

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. Here, 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 $l \geq k$; and the $\overset{\rightarrow}{=}$ symbol indicates equivalency, with a preferred direction to be used as a rewrite rule in our tool.

• $r[+]$	$\overset{\rightarrow}{=}$	$r\,;r[*]$	• $b[{-}{>}]$	$\overset{\rightarrow}{=}$	$\{(\sim b)[*]\,;\,b\}$
• $r[*k]$	$\overset{\rightarrow}{=}$	$r\,;r\,;\ldots;r$ (k times)	• $b[{-}{>}\,k]$	$\overset{\rightarrow}{=}$	$\{b[{-}{>}]\}[*k]$
• $r[*i{:}j]$	$\overset{\rightarrow}{=}$	$r[*i] \mid \ldots \mid r[*j]$	• $b[{-}{>}\,k{:}l]$	$\overset{\rightarrow}{=}$	$\{b[{-}{>}]\}[*k{:}l]$

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

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, and are formed from SEREs: $s := \{r\}$. SEREs and sequences are different entities, and production rules are more constrained than what was stated previously. Since we are mainly concerned with the effect of an operator, the exact syntax rules are deferred to IEEE Std. 1850 [2005].

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), and are presented with the simple subset modifications (Section 4.4.4 in IEEE Std. 1850 [2005]). 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. Operators next, next!, always, eventually!, until! and until are equivalent to standard LTL operators X, $X!$, G, F, U and W, respectively, and are omitted. Furthermore, as indicated in the Working Group issues for the next version of PSL, the until_ and until!_ operators' left-hand side arguments do not need to be restricted to Boolean expressions in the simple subset.

Definition 2.2. 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 i, j are nonnegative integers and k, l are positive integers with $j \geq i$ and $l \geq k$, then PSL *foundation language properties* are defined as follows in the simple subset.

• b	• (p)	• s
• $!b$	• p abort b	• $s!$
• p_1 && p_2	• b_1 <–> b_2	• s |–> p
• b || p	• b –> p	• s |=> p
• always p	• never s	• eventually! s
• p until b	• p until_ b	
• p until! b	• b_1 before b_2	• b_1 before_ b_2
• p until!_ b	• b_1 before! b_2	• b_1 before!_ b_2
• next p	• next_event (b) (p)	• next_a$[i:j](p)$
• next! p	• next_event! (b) (p)	• next_a!$[i:j](p)$
• next$[i](p)$	• next_event $(b)[k](p)$	• next_e$[i:j](b)$
• next!$[i](p)$	• next_event! $(b)[k](p)$	• next_e!$[i:j](b)$
• next_event_a$(b)[k:l](p)$	• next_event_e $(b_1)[k:l](b_2)$	
• next_event_a!$(b)[k:l](p)$	• next_event_e! $(b_1)[k:l](b_2)$	

This definition shows that Booleans and sequences can be used directly as properties, thereby indicating that the sequence or Boolean expression is expected to be matched, and that a nonoccurrence constitutes a failure of the property. The matching is weak, meaning that if the end of execution occurs before the matching is complete, then the property holds. A sequence can be made to be a strong sequence using the ! operator, thereby specifying not only that the sequence should be matched, but that it should be matched before the end of execution. The abort operator can be used to release an obligation on a property when a given Boolean condition occurs.

In the simple subset, negation and equivalency of full properties are not allowed, and must be performed only with Booleans. Property implication and property disjunction allow at most one of the arguments to be a property. In the case of the implication operator, the antecedent must be a Boolean. If the antecedent of the implication occurs, then the consequent property is expected to hold. A behavior similar to implication also exists for disjunction. If the Boolean is false then the argument property is expected to hold, and if the Boolean is true then the property holds. In the definition of properties, even though the Boolean is shown as the left-side argument of ||, reversed arguments are also acceptable. Property conjunction &&, not to be confused with SERE intersection, is used to specify that two properties must both hold.

The always and never operators specify how their argument property and sequence, respectively, should behave. The next operator starts the checking of its property argument in the cycle following its own activation. This is a weak property, meaning that if the next cycle does not occur, then the property holds. The strong version of this operator, namely next!, does not allow the end of execution to occur in the next cycle. In other words, the next cycle must be a valid execution cycle and the argument property must hold, in order for the next property to hold.

The until family of properties cause the continual checking of their argument property until the releasing Boolean occurs. In the overlapped versions (with the _), the argument property is also checked in the clock cycle were the Boolean occurs. In the strong versions (with the !), the Boolean must occur before the end of execution. The before family of operators specify that the left side Boolean should occur before the right side Boolean, or else the property fails. Overlapped and strong versions of this operator are also defined. The eventually! property states that its argument sequence must be observed before the end of execution occurs.

The next[]() properties are extensions of the next properties mentioned previously, with a parameter for specifying the n^{th} next cycle. This applies to both the weak and the strong versions. The next_a properties cause the checking of their argument property in a range of next cycles, specified with a lower and upper bound integer. The next_e properties apply only to Booleans, and are used to indicate that the given Boolean must be observed at least once within a specified range of next clock cycles.

So far, eight variations of next properties have been encountered that use the clock cycle as a basic unit. The remaining eight next-type properties in Definition 2.2 are based on a different unit, namely the next *event*. For example, next_event_a! is used to specify that an argument property must be true within a range of next occurrences of an argument Boolean, and that all next occurrences of the Boolean that were specified in the range must occur before the end of execution. A subtlety worth mentioning is that the next event of a Boolean can be in the current cycle.

The two forms of temporal implications (|–> and |=>) are referred to as overlapped and nonoverlapped suffix implication, respectively. In overlapped suffix implication, for every matching of the antecedent sequence, the consequent property is expected to hold. The property must hold starting in the last cycles of all the antecedent sequence's matches. In the nonoverlapped suffix implication, the consequent property is expected to hold starting in all cycles that follow any successful antecedent sequence match.

Properties that appear below the separating line in Definition 2.2 are those for which we will devise rewrite rules to the base cases (previous the line). Following the definition of properties, it can be observed that sequences and Boolean expressions can be interpreted in two modes in dynamic verification.

Definition 2.3. *Obligation mode*. Subexpression semantic 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.

Definition 2.4. *Conditional mode*. Semantic 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.

For example, in assertions

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

Table I. Conventional Automata vs. Sequence Automata

Operation	**Conventional Automata**	**Sequence Automata**
O1:	DETERMINIZE()	WEAKDETERMINIZE()
O2:	–	STRONGDETERMINIZE()
O3:	Union $+$ (also denoted $\cup$)	Disjunction $\vert$
O4:	Closure $*$	Unbounded repetition $[*]$
O5:	Concatenation *juxtaposition*	Concatenation ;
O6:	Intersection $\cap$	Length matching intersection &&
O7:	–	Fusion :
O8:	–	FIRSTFAIL()
Edge labels:	Distinct symbols	Boolean expressions

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 $(b_1 \texttt{->} \{b_2 ; b_3\})$;

both sequences are in *obligation* mode because their failure to occur is used to trigger a condition. When such cases occur, a start condition indicates that the sequences must occur, and that a nonoccurrence constitutes a violation of the expected behavior. This is in contrast to their use in the conditional mode, where they are used in a more direct pattern detection manner. The use of properties is illustrated in Example 2.5.

Example 2.5. Property used in the verification of bus arbiters:

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

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. In this example, a checker generator would be used to create a monitoring circuit for the assertion, for inclusion into hardware emulation, silicon debug or post-fabrication diagnosis. Special considerations such as assertion multiplexing and assertion grouping must be made when a *large* amount of checkers are to be emulated and probed.

2.1 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. In our case, the transition conditions are different than in conventional automata [Hopcroft et al. 2000]. The procedures for constructing *Sequence Automata* appears in Boulé and Zilic [2007], but will be reviewed further on to keep the paper self-contained.

Table I shows the differences and similarities between conventional automata and our sequence automata. Edge labels in conventional automata represent distinct, mutually exclusive symbols, while in sequence automata, they represent complete Boolean-layer expressions that are not necessarily mutually exclusive: a number of separate expressions can simultaneously evaluate to true. This creates a Nondeterministic Finite Automaton (NFA) because a given state may transition to more than one successor state for the same input. Conventional automata are nondeterministic when more than one outgoing edge from a state carries the same symbol. This fundamental difference between both automata models influences many operations and operators, not the least of which is DETERMINIZE().

In operation O1, the typical determinization procedure based on subset construction (Hopcroft et al. [2000], p. 61) is not strong enough to determinize sequence automata. Classical determinization operates only at the symbol level and does not take into account the Boolean expressions behind symbols. Hence, this is referred to as weak determinization in sequence automata, and is used extensively in our minimization procedure. The strong determinization in O2 is performed by an algorithm that is also based on subset construction, with added features that take into account the simultaneous nature of the underlying Boolean expressions on edges.

A similarity between both models, however, is the inductive construction procedure of automata (Hopcroft et al. [2000], p.103), also called McNaughton-Yamada NFA construction. As an example, the union of two automata is typically performed by adding a separate start state and activating both automata through ϵ edges. In our work, the operators O3, O4, and O5 for disjunction, unbounded repetition and concatenation are instead implemented using distinct automata algorithms, given that we do not use ϵ transitions. These algorithms produce automata that are equivalent to using the traditional approach followed by application of an ϵ-removal algorithm.

Length-matching intersection is affected by the Boolean nature of edge labels, and must be modified from its conventional version. Implementing the O6 intersection operator in both cases is done using product construction (Hopcroft et al. [2000], p.135); however, in the case of sequence automata, the condition on equality of symbols must be relaxed, given their Boolean nature. The algorithm must allow for added conditions in which edges from each argument are simultaneously true, even if they represent different expressions.

The fusion in O7 is not an operator typically encountered in regular expressions, but is part of PSL's sequential regular expressions. Since fusion is an overlapped concatenation, a separate fusion algorithm was designed to merge the two argument automata by adding fusion edges between them. The fusion edges are intersection edges from all the combinations of edges that hit final states in the first automaton and edges that leave the start state in the second automaton.

Henceforth, $\mathcal{A}(s)$ and $\mathcal{A}(b)$ denote *obligation mode* automata for sequences and Boolean expressions in accordance with Definition 2.3. This mode will be employed when Boolean expressions or sequences 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 $\mathcal{A}_C(s)$ and $\mathcal{A}_C(b)$. Since a Boolean b can be seen as the sequence $\{b\}$, the construction of automata for Booleans in both modes is subsumed by the construction of automata for sequences.

The entire procedure described thus far creates conditional mode automata $\mathcal{A}_C()$ for occurrence detection, and are consistent with Definition 2.4. Creating

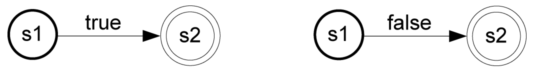

Fig. 1. a) Automaton for $\mathcal{A}_C(true)$, $\mathcal{A}(false)$; b) Automaton for $\mathcal{A}_C(false)$, $\mathcal{A}(true)$.

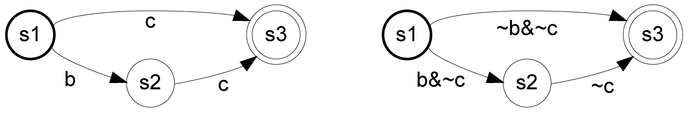

Fig. 2. a) Automaton for $\mathcal{A}_C(\{b[{*}0{:}1]\,;c\})$; b) Automaton for $\mathcal{A}(\{b[{*}0{:}1]\,;c\})$.

an obligation mode automaton $\mathcal{A}()$ involves applying the FirstFail() procedure to a conditional mode automaton as follows:

$$\mathcal{A}(s) = \text{FirstFail}(\mathcal{A}_C(s))$$

As its name implies, the FirstFail() function (O8) is used to transform the argument automaton to detect the first noncompletion of its equivalent PSL expression, for a given start condition. This function makes use of the strong determinization function mentioned previous, along with other edge manipulations.

The way sequence automata (which include Boolean automata) are used to form properties is at the core of this article. As will be presented in the next section, a property automaton can be used directly as an assertion automaton. In an assertion automaton, an assertion violation is reported each time a final state is activated. For a given assignment of various Boolean labels, all conditions that are true will cause a transition into a new *set* of active states.

PSL does not have a prescribed operational semantics [Claessen and Martensson 2004], and alternative runtime interpretations are possible. To provide useful debug information, we designed our automata algorithms such that the assertion result signal provides a continuous report of when the assertion has failed, rather than simply indicate a yes/no answer obtained at the end of execution. Event-driven simulators such as Mentor Graphics' Modelsim may have different semantics, as their interpretation of PSL involves a very different process.

Figures 1(a) and 1(b) show simple automata for the Boolean expressions *true* and *false*, in both modes. A generalized Boolean automaton is identical to these, with the corresponding Boolean expression inscribed on the edge label. In Figure 1(b), since the *false* symbol can never be true, the automaton never reaches the final state (double circle). When a conditional mode automaton reaches a final state, 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 bold is the start state of the automaton, and for a top-level automaton, it is the only active state when reset is released. Figure 2(a) shows a conditional mode automaton for detecting the sequence $\{b[{*}0{:}1]\,;c\}$. Figure 2(b) shows how the same sequence is processed in obligation mode by an automaton.

3. TRANSFORMING PROPERTIES INTO CIRCUITS

We now show how to transform properties into automata, for subsequent conversion to circuits. 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 originates. The signal that 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). In sum, automata are implemented using combinational logic and flip-flops, and do not have to be deterministic. The conversion of automata to circuits is illustrated in a summary example further in this section.

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 subproperty rooted at that node. The parent then builds its own subproperty automaton from its children automaton(s), using a variety of transformations and operations. This is referred to as *Recursive Mechanism #1*.

Properties are inherently in obligation mode; however, Definition 2.3 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 is modeled as:

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

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

In our work, a full automaton approach allows the production of efficient automata. Consider the following examples, for which a modular approach will generate more checker code for the assertion on the left, even though in dynamic verification both assertions are semantically identical.

$$\text{assert never } \{b[*1{:}2]\,;c\}; \qquad \text{assert never } \{b\,;c\};$$

3.1 Implementation of Base Cases

The approach for implementing the base cases consists in taking the automaton(s) returned by the argument(s) 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 automata implementations of the base cases are shown in Table II.

Parentheses affect only the syntax tree and can be dropped as shown in the implementation of base case B1. Certain properties are relegated to the Boolean layer in the simple subset, such as in cases B2 and B3. The negation and equivalency of properties—as allowed in full PSL—create properties that are not suitable for monotonically advancing time. When used as properties, the

Table II. Base Cases' Automaton Implementation

	Property	Automaton Implement.	Comment
B1:	(p)	$\mathcal{A}(p)$	Parentheses used only for grouping
B2:	$!b$	$\mathcal{A}(\sim b)$	Simple automaton for Boolean
B3:	b_1 <-> b_2	$\mathcal{A}((\sim b_1 \mid b_2)$ & $(\sim b_2 \mid b_1))$	Simple automaton for Boolean
B4:	b	$\mathcal{A}(b)$	Obligation mode Boolean
B5:	s	$\mathcal{A}(s)$	Obligation mode sequence
B6:	$s!$	MAKESTRONG($\mathcal{A}(s)$)	Add EOE edges
B7:	p abort b	ADDLITERAL($\mathcal{A}(p)$, $\sim b$)	Add literal to all edge symbols
B8:	p_1 && p_2	$\mathcal{A}(p_1) \mid \mathcal{A}(p_2)$	Both properties get activation
B9:	s |-> p	$\mathcal{A}_C(s) : \mathcal{A}(p)$	Uses automata fusion

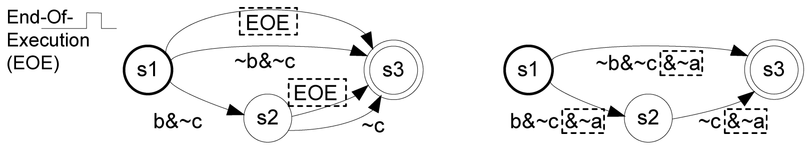

Fig. 3. a) MAKESTRONG($\mathcal{A}(\{b[*0{:}1]\,;c\})$); b) ADDLITERAL($\mathcal{A}(\{b[*0{:}1]\,;c\})$, $\sim a$).

automata for Booleans and sequences are built in obligation mode, as shown in cases B4 and B5. Implementing strong sequences involves constructing the obligation mode automaton with a slight modification, as shown in B6. The MAKESTRONG() function adds edges that cause the automaton to transition from any active state to a final state upon activation of the End-Of-Execution (EOE) signal. If the automaton is processing a sequence when the EOE occurs, an error is detected. For example, applying the function to the automaton in Figure 2(b) yields the automaton in Figure 3(a).

Handling the abort property also involves modifying the automaton of the property argument, as shown in case B7. When the abort operator is encountered in the syntax tree, the argument property's automaton is built and a new primary symbol for the abort condition's Boolean is created. The function ADDLITERAL() then adds a literal (a conjunct) to each edge symbol in the property automaton such that when the abort condition is asserted, all transitions in the automaton are disabled and the automaton is reset. The added literal corresponds to the negation of the abort Boolean, given the conjunctions with existing symbols. For example, aborting the automaton in Figure 2(b) yields the automaton in Figure 3(b).

Property conjunction (B8) is implemented using the | operator, which represents automata union (disjunction). The disjunction is required because a failure by either subproperty is a failure for the && property. Both argument automata are simultaneously activated by the parent node's subautomaton in the syntax tree, and when either one reaches a final state, a failure is detected. The automata disjunction is the same as in SERE disjunction, where both automata are concurrently searching for a match, for a given start condition (activation).

Table III. Rewrite Rules Based on Property Implication

R1:	$b \Vert p$	$\overset{\rightarrow}{=}$	$\{\sim b\} \ \vert\!-\!> p$	R4:	always p	$\overset{\rightarrow}{=}$	$\{[+]\} \ \vert\!-\!> p$
R2:	$b -> p$	$\overset{\rightarrow}{=}$	$\{b\} \ \vert\!-\!> p$	R5:	never s	$\overset{\rightarrow}{=}$	$\{[+] : s\} \ \vert\!-\!> \mathit{false}$
R3:	$s \ \vert\!=\!> p$	$\overset{\rightarrow}{=}$	$\{s \ ; \mathit{true}\} \ \vert\!-\!> p$	R6:	p until b	$\overset{\rightarrow}{=}$	$\{(\sim b)[+]\} \ \vert\!-\!> p$
R7:	p until_ b	$\overset{\rightarrow}{=}$	$\{\,\{(\sim b)[+]\} \vert \{b[->]\}\,\} \ \vert\!-\!> p$				
R8:	next_event_a$(b)[k{:}l](p)$	$\overset{\rightarrow}{=}$	$\{b[->k{:}l]\} \ \vert\!-\!> p$				

Overlapped suffix implication (also called property implication) is implemented using a conditional mode automaton for the antecedent sequence, which is then fused with the consequent property, as shown in case B9. When used in the context of automata, the ":" symbol denotes the actual automata fusion algorithm, the same one that is invoked when the SERE fusion operator ":" is encountered. The fusion algorithm avoids building fusion edges containing the EOE symbol such that activations (antecedents) occurring at the end of execution do not cause a failure.

Using fusion in properties does not create unwanted side effects, as 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 consistent with the formal semantics of PSL in Appendix B in IEEE Std. 1850 [2005]. As an example, when a sequence automaton's start state is a final state, and this sequence is used as an antecedent in suffix implication, the empty match can not cause the consequent to be enforced. When a conditional mode sequence automaton is used at the property level, the start state is made nonfinal. When an obligation mode sequence automaton is built for a sequence that cannot hold, the automaton from Figure 1(a) is returned to the parent.

3.2 Rewrite Rules for Properties

Most properties from Definition 2.2 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 rewritten when encountered during checker generation. The rules by which these properties are rewritten, that is, *rewrite rules*, are categorized in three groups appearing in Tables III to V.

Using the sugaring definitions from Appendix B in IEEE Std. 1850 [2005] as rewrite rules is generally 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.

In the simple subset, one of the properties used in disjunction must be Boolean (for simplicity the Boolean expression is shown as the left argument). The R1 rewrite rule is based on the fact that if the Boolean expression is not true, 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 nonoverlapped property implication follows from its sugaring definition in Appendix B in IEEE Std. 1850 [2005]. The simple subset

Table IV. Rewrite Rules Based on Using Sequences as Properties

R9:	b_1 before b_2	$\overset{\rightarrow}{=}$	$\{(\sim b_1 \& \sim b_2)[*]\ ;\ (b_1 \& \sim b_2)\}$
R10:	b_1 before! b_2	$\overset{\rightarrow}{=}$	$\{(\sim b_1 \& \sim b_2)[*]\ ;\ (b_1 \& \sim b_2)\}!$
R11:	b_1 before_ b_2	$\overset{\rightarrow}{=}$	$\{(\sim b_1 \& \sim b_2)[*]\ ;\ b_1\}$
R12:	b_1 before!_ b_2	$\overset{\rightarrow}{=}$	$\{(\sim b_1 \& \sim b_2)[*]\ ;\ b_1\}!$
R13:	next_event_e$(b_1)[k{:}l](b_2)$	$\overset{\rightarrow}{=}$	$\{b_1[->k{:}l]\ :\ b_2\}$
R14:	next_event_e!$(b_1)[k{:}l](b_2)$	$\overset{\rightarrow}{=}$	$\{b_1[->k{:}l]\ :\ b_2\}!$

does not affect this definition; therefore, it can be used directly as a rewrite rule.

As explained in the previous section, suffix implication has a conditional-mode sequence as an antecedent, and a property as a consequent. When a property must always be true (R4), it can be seen as the consequent of a suffix implication with a perpetual start condition ([+] is sugaring for *true*[+]). When a sequence must not occur (R5), a property that fails instantly is triggered upon detection of the sequence. Because overlapped suffix implication does not have a clock cycle delay between antecedent and consequent, these rewrites offer the correct timing.

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 a start condition to p for each cycle of consecutive $\sim b$'s. In our interpretation of operational semantics for the until operator, the property is allowed to fail multiple times for a given start condition 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 a start condition to the property each time b occurs within the specified range after the current property received its start condition.

The before family of properties in Table IV (R9 to R12) can be modeled by obligation mode sequences. The overlapped versions state that b_1 must be asserted before or simultaneously with b_2. The next_event_e properties (R13 and R14) 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 versions of these properties are created by using strong sequences.

In Table V, the strong versions of the until properties (R15 and R16) are created by using the weak versions, 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, the assertion will trigger, even though the weak until may have always held.

The R17 to R20 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

Table V. Rewrite Rules Based on Property Variations

R15:	p until! b	$\overset{\rightarrow}{=}$	(p until b) && ({b[–>]}!)
R16:	p until!_ b	$\overset{\rightarrow}{=}$	(p until_ b) && ({b[–>]}!)
R17:	next p	$\overset{\rightarrow}{=}$	next[1](p)
R18:	next! p	$\overset{\rightarrow}{=}$	next![1](p)
R19:	next_event(b)(p)	$\overset{\rightarrow}{=}$	next_event(b)[1](p)
R20:	next_event!(b)(p)	$\overset{\rightarrow}{=}$	next_event!(b)[1](p)
R21:	next[i](p)	$\overset{\rightarrow}{=}$	next_event($true$)[i +1](p)
R22:	next![i](p)	$\overset{\rightarrow}{=}$	next_event!($true$)[i +1](p)
R23:	next_a[i:j](p)	$\overset{\rightarrow}{=}$	next_event_a($true$)[i+1 : j+1](p)
R24:	next_a![i:j](p)	$\overset{\rightarrow}{=}$	next_event_a!($true$)[i+1 : j+1](p)
R25:	next_e[i:j](b)	$\overset{\rightarrow}{=}$	next_event_e($true$)[i+1 : j+1](b)
R26:	next_e![i:j](b)	$\overset{\rightarrow}{=}$	next_event_e!($true$)[i+1 : j+1](b)
R27:	next_event(b)[k](p)	$\overset{\rightarrow}{=}$	next_event_a(b)[k:k](p)
R28:	next_event!(b)[k](p)	$\overset{\rightarrow}{=}$	next_event_a!(b)[k:k](p)
R29:	next_event_a!(b)[k:l](p)	$\overset{\rightarrow}{=}$	next_event_a(b)[k:l](p) && {b[–>l]}!

1 is implicit. Since the right-hand sides of these 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.

The family of rules in R21 to R26 are 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. 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)[k](p) requires a positive count for k. Incidentally, next[0](p) is equivalent to (p).

The strategy behind the R27 and R28 rewrites is to utilize the next_event_a form, with identical upper and lower bounds for the range. Rule R29 handles the strong version of the full next_event_a property. Similarly to the strong nonoverlapped 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 before execution terminates.

Example 3.1. Summary example for rewrite rules and base cases:

assert always {∼a ; a} |–> {b[*0:1] ; c};

The subautomaton for the left side of the temporal implication is a three state conditional mode automaton with two edges, and is visible in the left side of Figure 4(a). The subautomaton for the right side of the implication appears in the right side of the same figure, and was also shown in Figure 2(b). The suffix implication is implemented with automata fusion and thus merges these two subautomata together, as shown in the right side of Figure 4(b). In the left side of Figure 4(b), the antecedent of the always rewrite is prepared. In Figure 4(c) the suffix implication contained in the always rewrite is also implemented with

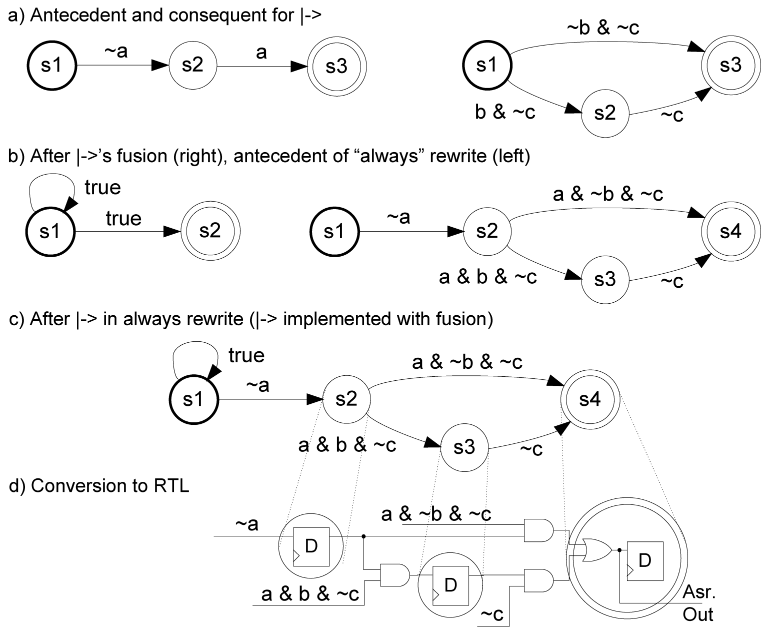

Fig. 4. Complete checker for assertion example: assert always $\{{\sim}a\ ;\ a\}$ |-> $\{b[{*}0{:}1]\ ;\ c\}$;.

fusion, and the resulting minimized automaton is shown. Converting the automaton to RTL is done as indicated at the beginning of Section 3, and appears in Figure 4(d). The start state is optimized given that the true self loop perpetually keeps the state active. The checker's output signal can then be monitored during device execution to locate errors.

3.3 A Special Case for eventually!

Compilation of a PSL property normally involves recursively scanning the syntax tree of the PSL expression, whereby each node returns an automaton for the sub-property rooted at that node. The parent builds its own subproperty automaton from its children automata; this was referred to as recursive mechanism #1. In an alternate mechanism (#2), a parent sends a precondition in automaton form to a child, whereby the *child* node is responsible for building the subautomaton (with its activations) and returning it to the parent. If other child nodes exist, the parent forms other precondition automata, possibly using the automata returned by previous child nodes. When finished, the parent returns an automaton formed from the children automata (directly or with modifications) to its parent. This recursive process continues until the top-level directive's automaton is formed. Recursive mechanism #2 is a prerequisite for automata splitting. Automata splitting is required for assertion threading and activity monitors [Boulé et al. 2006], and also for the more efficient form of eventually!.

In Boulé and Zilic [2006], rewriting eventually! is based on the following rule

$$\text{eventually! } s \overset{\rightarrow}{=} \{[{*}]{:}s\}!$$

This rule has the advantage of preserving the full automaton approach; however, given that the sequence in the right-hand side of the rule is in obligation mode, it is not the most efficient form. This subsection details the use of a more efficient procedure for implementing the eventually! property, for use when automata splitting is allowed and the use of separate logic and wire declarations are permitted. An efficient implementation of eventually! is also important for the cover directive which is often rewritten to the eventually! operator [Ziv 2003]. Although automata optimizations can no longer cross split automata boundaries, in the case of eventually! the improvements brought upon by the split approach outweigh this disadvantage.

In the split approach, implementing the "eventually! *s*" property is done with a conditional-mode automaton. When the conditional-mode automaton for sequence *s* is built, its start state is made nonfinal. At this point, if the sequence automaton has no final states, the sequence cannot eventually occur, and the failure must be signaled at the end of execution. In this case an automaton for "never {*EOE*}" is returned to the parent node in the syntax tree of the assertion.

If the sequence automaton is not degenerate, then a more complex algorithm is applied. First, any incoming edges to the start state are removed. Then any outgoing edges from the final states are removed. The automaton must be implemented as a module, for which a result signal is declared. The result signal is then used, complemented, as the symbol of a self loop on the start state. This has the effect of keeping the start state active until at least one occurrence of the sequence has manifested itself. The actual result signal of the eventually! operator therefore corresponds to the output of the start state's flip-flop. In this manner, no extra states (hence flip-flops) are used. The actual result signal is implemented in automaton form before being returned to eventually!'s parent node. This consists in preparing a two-state automaton whereby the second state is a final state, the start state has a self *true* loop, and an edge from the start state to the second state carries a symbol corresponding to the result signal.

When a precondition automaton is passed to eventually! in the recursive compilation, the precondition automaton is implemented as a module, for which a result signal is declared. This signal constitutes the precondition signal for the eventually! automaton. Each time the precondition is asserted the conditional mode automaton for eventually! is flushed, with the start and final state's edges modified as described previously. Automaton *flushing* consists in deactivating the edges for all states except the start state. This is accomplished by and-ing a negated literal of the precondition signal to each outgoing edge symbol of each nonstart state. In this manner, each new precondition guarantees a new complete obligation. The precondition automaton used in this technique implies that recursive mechanism #2 must be employed.

Figure 5 shows an example of the efficient eventually! strategy. The property is actually implemented as two automata, and the automaton at the top right in the figure is returned by the property's node in the syntax tree. Since the always property is the argument of the assert directive, the returned automaton is directly implemented in RTL. The grey state also serves as the memory state, which is deactivated once the obligation is fulfilled (sequence occurred).

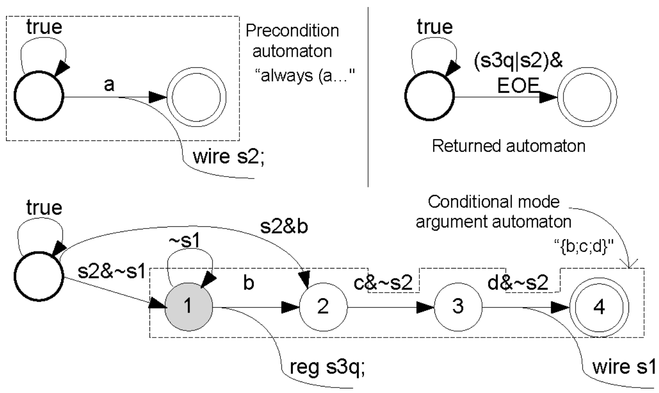

Fig. 5. Automata Splitting for eventually!; ex: always (a –> eventually! {b;c;d}).

Automaton flushing is also visible with the added "$\sim s2$" literals. If the always property was part of a more complex assertion, the returned automaton would be used by the parent property to continue to build the complete automaton for the assertion. Automata splitting and the use of custom logic (an inverter) could also be used for efficient automata negation by avoiding the determinization step.

4. EXPERIMENTAL RESULTS

The circuits produced by the MBAC checker generator are evaluated using various test assertions and measured against the only comparable tool available to us. The hardware comparison metrics involve synthesizing the assertion circuits using ISE 8.1.03i 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. Assertion signals are sampled by a FF in both tools to achieve the same timing for simulations and equivalence checking.

The assertions used for evaluating checker generators typically do not contain complex Boolean expressions because such expressions have no effect on the temporal aspect of assertion checkers. Consequently, 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 Cohen et al. [2004]) 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 VI, whereas the synthesis results are reported in Table VII. Properties P18 and P20–P24 are from [Borrione et al. 2005]. For cases P1 to P12 and P16, no synthesis results are given because the properties are not supported by FoCs. Property P14 exceeded the internal limits in FoCs and no output was produced.

With the exception of P13, we observed that when FoCs produced a checker, both tools produce functionally equivalent checkers. Functional equivalence

Table VI. Benchmarking Properties

Property (“assert Px;”, where Px is:)			
P1	always ({a;d} |–> next_e[2:4](b)) until c		
P2	always ({a;b} |–> eventually! {c;d}) abort e		
P3	always {a;b[*0:2];c} |=> ({d[*2]} |–> next ∼e)		
P4	always (a –> ((eventually! b[*5]) abort c)) abort d		
P5	always ((a –> (b before c)) && (c –> eventually! {b;d})) abort e		
P6	always {a;b;c} |=> never {d[*0:3];e}	P7	always a –> next_a![2:4](b)
P8	always (a –> {[*0:7];b}) abort ∼c	P9	always a –> next_e![2:4](b)
P10	always a –> next_event_e!(b)[2:4](c)	P11	always a –> ({b;c} until! d)
P12	always a –> next_event_a!(b)[5:10](c)	P13	always a –> (b until! c)
P14	always {a} |–> {{b;c[*]}:{d[*];e;f}}	P15	never {a;[*];{b;c}[+]}
P16	always (e || (a –> ({b;c} until d)))	P17	always a –> (b before!_ c)
P18	always (a –> next_event_e(b)[1:6](c))	P19	always a –> eventually! b
P20	always (a –> next (next_a[2:10](next_event(b)[10]((next_e[1:5](d)) until (c)))))		
P21	always ((a –> next(next[10](next_event(b)((next_e[1:5](d)) until (c))))) || e)		
P22	(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))))))		
P23	always (a –> (next_event_a(b)[1:4](next((d before e) until (c)))))		
P24	always (a –> (next_event(c)((next_event_e(d)[2:5](e)) until (b))))		

Table VII. Hardware Metrics for Checkers (P1-P12 are not supported by FoCs yet)

	MBAC				**MBAC**			**FoCs**		
Px	FF	LUT	MHz	**Px**	FF	LUT	MHz	FF	LUT	MHz
P1	**6**	**5**	**611**	P13	**2**	**3**	**612**	3	4	474
P2	**4**	**6**	**470**	P14	**8**	**23**	**295**	No Output		
P3	**7**	**6**	**611**	P15	**3**	**2**	**611**	4	3	564
P4	**6**	**9**	**460**	P16	**3**	**5**	**469**	Not Supported Yet		
P5	**4**	**7**	**473**	P17	**2**	**3**	**610**	3	4	474
P6	**7**	**7**	**419**	P18	**7**	**8**	445	12	12	**564**
P7	**5**	**2**	**445**	P19	2	2	564	2	2	564
P8	**8**	**8**	**667**	P20	**26**	**10**	**375**	200	106	299
P9	**5**	**4**	**433**	P21	**17**	**7**	564	23	13	564
P10	**5**	**6**	**456**	P22	**23**	**11**	**513**	46	43	312
P11	**3**	**4**	**505**	P23	**7**	**5**	**552**	24	24	311
P12	**11**	**7**	**326**	P24	**8**	**11**	**439**	40	40	408

was formally verified using the Cadence SMV model checker, to which we input the Verilog code obtained from the two checker generators. For a given assertion, the checkers generated by both tools are instantiated and a property stating the equivalency of their outputs is specified using the xor gate approach (i.e., the xor between the two functions must always be false). Test cases P20 and P24 exceeded the maximum memory capacity of the model checker, and were compared in simulation instead. The FoCs checkers for those two cases have 309 and 175 state elements respectively in the HDL code, versus 27 and 9 respectively in MBAC’s checkers.

Table VIII. Hardware metrics for Implementations of eventually!

Property	**Splitting**			**Rewrite**		
(F ≡ eventually!, G ≡ always)	FF	LUT	MHz	FF	LUT	MHz
eventually! {b;c;d}	**4**	**4**	**559**	5	11	388
eventually! {a;b;c;d;e}	**6**	**6**	**559**	17	52	237
always (a–> eventually! {b;c;d})	**4**	**4**	**559**	5	10	395
G (a–> eventually! {a;b;c;d;e})	**6**	**5**	**564**	17	44	236
G (a–> eventually! {b[*5:10]})	6	6	**559**	6	**5**	548
G (a–> F {b; {c[*0:2]} | {d[*0:2]} ; e})	**6**	**8**	**444**	7	20	329
G (a–> F { {{c[*1:2];d}[+]} | {e[–>2]} })	6	**8**	**434**	**5**	9	392
G (a–> F {{{c[*1:2];d}[+]}:{{e[–>]};d}})	6	7	**417**	**5**	7	395
G (a–> F {{{c;d}[*]}&&{∼e[*];e;∼e[*]}})	4	**5**	**474**	4	7	465
G ({a;b[*0:2];c} |–> F {d;e[*1:3];f})	**9**	**9**	**473**	10	17	309
G (a–> F { {{b;c[*1:2];d}[+]} && {b;{e[–>2:3]};d} })	**17**	**27**	**395**	68	245	207

Test cases P20 and P24 were compared using a testbench of 10^5 biased pseudo-random test vectors. For each assertion, the circuits produced by both tools offer the same behavior on every clock cycle. In biased random vector generation, signal probabilities are adjusted in order for the assertions to trigger reasonably often. This method is not a proof that the circuits are functionally equivalent; however, combined with the fact that model checking produced no counterexample before reaching its limit, this does offer reasonable assurance.

For test case P13, slight differences in behavior were noticed due to the operational semantics of "*p* until *b*," where 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.

With P17 and P19, the behavior of the checkers is identical between both tools, and only a slight difference occurs when the End-Of-Execution (EOE) signal activates. This was alleviated by using a monostable flip-flop on the assertion outputs such that when the assertion triggers, the output remains in the triggered state. This was used because the semantics of the checkers does not have to be defined after the EOE occurs. In all test cases, our circuits are more resource-efficient than FoCs'.

Table VIII shows the advantages of the split-automata method in the implementation of eventually!, compared to the rewrite rule from [Boulé and Zilic 2006]. The split-automata method scales much better because a conditional mode automaton can be employed as opposed to an obligation mode automaton, which can be exponentially larger given the required strong determinization [Boulé and Zilic 2007]. In the test cases, the split-automata method produces faster circuits, and except for a few small examples, requires less hardware. In all eleven test cases, functional equivalence of the checkers was formally verified by model checking. These examples show that in general not all sequential optimizations can be performed by traditional synthesis tools, and efforts to optimize the checkers pre-synthesis should always be made.

5. CONCLUSIONS AND FUTURE WORK

We have presented methods for implementing PSL properties in checker generators. The base cases were handled using a variety of automata techniques. A set of rewrite rules that account for all peculiarities of PSL operators were also devised. These rewrites represent the simplest way to support such operators in the kernel of the checker generator, and can be implemented in any tool that utilizes PSL for dynamic verification. A new recursive mechanism was developed, and along with automata splitting, a more resource-efficient implementation of the eventually! operator was introduced. Experimental results show the effects of this method, and also show the overall strength of the MBAC generator, with respect to functionality and the size of the generated assertion circuits. We have also presented in a single paper the only (to our knowledge) checker implementation of all PSL operators, both strong and weak, with all variants of operator families.

Many of the techniques developed in this paper also apply to SVA assertions. For example, the disable iff operator in SVA is very similar to PSL's abort operator, and both forms of suffix implications are identical in both languages. Property conjunction and disjunction are also similar; however, always and never have slight differences that should be straightforward to convert.

We plan to ad support for multiple clock domains in a single assertion, and to benefit from techniques such as in Morin-Allory and Borrione [2006b], to provide an independent theorem prover based verification of our rewrite rules.

REFERENCES

Abarbanel, Y., Beer, I., Glushovsky, L., Keidar, S., and Wolfsthal, Y. 2000. FoCs: Automatic generation of simulation checkers from formal specifications. *Conference on Computer Aided Verification*. 538–542.

Borrione, D., Liu, M., Morin-Allory, K., Ostier, P., and Fesquet, L. 2005. Online assertion-based verification with proven correct monitors. In *Proceedings of the 3rd ITI International Conference on Information & Communications Technology (ICICT)*. 123–143.

Boulé, M., Chenard, J., and Zilic, Z. 2006. Adding debug enhancements to assertion checkers for hardware emulation and silicon debug. In *Proceedings of the 24th IEEE International Conference on Computer Design (ICCD)*. 294–299.

Boulé, M. and Zilic, Z. 2006. Efficient automata-based assertion-checker synthesis of PSL properties. In *Proceedings of the IEEE International High Level Design Validation and Test Workshop (HLDVT)*. 69–76.

Boulé, M. and Zilic, Z. 2007. Efficient automata-based assertion-checker synthesis of SEREs for hardware emulation. In *Proceedings of the 12th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 324–329.

Claessen, K. and Martensson, J. 2004. An operational semantics for weak PSL. In *Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*. 337–351.

Cohen, B., Venkataramanan, S., and Kumari, A. 2004. *Using PSL/Sugar for formal and dynamic verification*. VhdlCohen Publishing, Los Angeles, CA.

Das, S., Mohanty, R., Dasgupta, P., and Chakrabarti, P. 2006. Synthesis of system verilog assertions. In *Proceedings of the Conference on Design Automation and Test in Europe (DATE)*. 70–75.

Foster, H., Krolnik, A., and Lacey, D. 2004. *Assertion-Based Design, 2nd Ed.* Kluwer Academic Publishers.

GHEORGHITA, S. AND GRIGORE, R. 2005. Constructing checkers from PSL properties. In *Proceedings of the 15th International Conference on Control Systems and Computer Science (CSCS) 2*, 757–762.

GORDON, M., HURD, J., AND SLIND, K. 2003. Executing the formal semantics of the accelera property specification language by mechanised theorem proving. Lecture Notes in Computer Science, vol. 2860, 200–215.

HOPCROFT, J., MOTWANI, R., AND ULLMAN, J. 2000. *Introduction to Automata Theory, Languages and Computation, 2nd Ed.* Addison-Wesley.

IBM ALPHAWORKS. 2006. FoCs Property Checkers Generator, version 2.04. www.alphaworks.ibm.com/tech/FoCs.

IEEE STD. 1850. 2005. IEEE Standard for Property Specification Language (PSL). Institute of Electrical and Electronic Engineers, Inc., New York, NY.

MORIN-ALLORY, K. AND BORRIONE, D. 2006a. Online monitoring of properties built on regular expression sequences. *Forum on Specification Design Languages (FDL)*.

MORIN-ALLORY, K. AND BORRIONE, D. 2006b. Proven correct monitors from PSL specifications. In *Proceedings of the Conference on Design Automation and Test in Europe (DATE)*. 1246–1251.

ZIV, A. 2003. Cross-product functional coverage measurement with temporal properties-based assertions. In *Proceedings of the Conference on Design Automation and Test in Europe (DATE)*, 834–839.

Received May 2006; revised May 2007; accepted July 2007