Efficient Automata-Based Assertion-Checker Synthesis of SEREs for Hardware Emulation

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

***Abstract*— In this paper, we present a method for generating checker circuits from sequential-extended regular expressions (SEREs). Such sequences form the core of increasingly-used Assertion-Based Verification (ABV) languages. A checker generator capable of transforming assertions into efficient circuits allows the adoption of ABV in hardware emulation. Towards that goal, we introduce the algorithms for *sequence fusion* and *length matching intersection*, two SERE operators that are not typically used over regular expressions. We also develop an algorithm for generating *failure detection automata*, a concept critical to extending regular expressions for ABV, as well as present our efficient symbol encoding. Experiments with complex sequences show that our tool outperforms the best known checker generator.**

I. Introduction

Hardware verification aims to ensure that a design fulfills its given specification by either formal or dynamic (simulation based) techniques. Assertion-Based Verification (ABV) is quickly emerging as the dominant methodology for performing hardware verification in practice. Assertions are statements added to the source code that specify how a design should behave. Hardware assertions are typically written in a verification language such as PSL (Property Specification Language) or SVA (SystemVerilog Assertions). In dynamic verification, a simulator can monitor the Device Under Verification (DUV) and report assertion violations.

As circuits become more complex, hardware emulation is becoming an increasingly important asset for verification. Hardware emulation achieves the traditional dynamic verification goals by loading and executing the circuit on a reprogrammable hardware substrate, typically using programmable logic devices. Once implemented in hardware, the emulator fully exploits the inherent circuit parallelism, as opposed to performing a serial computation in a simulation kernel.

A problem arises when performing ABV in a hardware emulation environment: assertions are usually expressed at higher levels of abstraction that are not easily expressed in a Hardware Description Language (HDL). Assertion languages allow compact representations of complex temporal relations between the circuit signals. In order to consolidate assertion-based verification and emulation, a checker generator is needed to generate hardware assertion checkers, which are meant to catch errors through assertion violations [1], [2]. Assertion checkers are RTL (Register Transfer Level) implementations of assertions that can be included in the DUV.

Assertion languages such as PSL and SVA make use of sequential-extended regular expressions (SEREs) for describing temporal sequences. This paper introduces the algorithms for transforming *SEREs* into resource efficient circuits suitable for hardware emulation, and demonstrates their applicability within our complete checker generator tool. Synthesizing *resource efficient* assertion checker circuits is crucial for emulation because the assertion checking circuits compete with the DUV for resources in an emulator. Bus interface designs that incorporate hundreds or even thousands of assertions are not uncommon. To our knowledge, the only other standalone tool capable of generating hardware checkers from PSL assertions is IBM's FoCs property checkers generator [1], [3]. It will be shown that the circuits generated by our tool can be significantly more efficient.

Generating hardware from regular expressions is of interest in applications such as protocol compilers [4] and Esterel hardware compilers [5], as well as in Production Based Specification [6]. The body of research on converting regular expressions into hardware from [4], [6], [7], [8] can be a starting point for the task of transforming sequences into hardware. However, the SEREs used in assertion languages are much more demanding. First, the symbols in sequences correspond to many simultaneously interacting Boolean expressions, as opposed to the plain symbols used in pattern matching. Second, the notion of sequence intersection, which is often not treated in regular expression matching, must be reconsidered given the first point, and must be of length-matching type. Third, while automata-based regular expression recognizers only require the matching (detection) of patterns, assertions often require explicit detection of sequence failures. Fourth, sequence fusion, an operator not found in conventional regular expressions, is required in assertions sequences.

The sequential-and operator from [6] consists in performing the Boolean conjunction of the *result* of sequences (i.e. in their final cycle), and does *not* equate to the proper length-matching intersection found in assertion languages. Furthermore, the sequential-not operator does not correspond to the type of negation required to perform sequence-failure detection, as the negation is performed on the *result* signal of a sequence.

The automata produced in [9], [10] can be used to check a property during simulation. These types of checkers are meant to indicate the status of the property at the end of execution only. This is not ideal for debugging purposes; it is more informative to provide a dynamic trace of the assertion and to report each instance where the assertion fails.

All the algorithms described in this paper are implemented in our PSL checker generator called MBAC. The basic framework for MBAC, introduced in [2], did not include the automata-based implementation of SERE checkers. By adding automata-based SERE recognition, checkers which contain temporally infinite sequences under all operators can be generated. The contributions of this paper are:

- Introduction of procedures for generating resource efficient sequence-checking circuits, with special attention paid to intersection and fusion, operators that are not typically used in regular expressions;
- Introduction of the algorithms used to convert sequence-checking automata into sequence-failure automata;
- Improvement upon the best known checker generator in terms of efficiency, capability and correct behavior of the generated circuits.

II. Background

A string is a sequence of symbols from an alphabet Σ, including an empty string, denoted as ϵ. A regular expression RE is a pattern that describes a set of strings, or a *language* of RE, denoted $L(RE)$. Regular Expressions and the corresponding languages are described as follows:

- a symbol s from Σ is a RE; $L(s) = \{s\}$
- ϵ is a RE; $L(\epsilon) = \{\epsilon\}$
- $r_1|r_2$ is a RE; $L(r_1|r_2) = L(r_1) \cup L(r_2)$ (set union)
- r_1r_2 is a RE; $L(r_1r_2) = L(r_1)L(r_2)$ (set concat.)
- r_1* is a RE; $L(r_1*) = (L(r_1))*$ (Kleene closure)

where r_1 and r_2 are regular expressions, and the Kleene closure (Kleene star) is an operator that creates the strings formed by concatenating zero or more strings from a language.

A regular expression's language can be expressed equivalently, in a form suitable for computation, by a finite automaton [11] that accepts the same language. A Finite Automaton A is described by a quintuple $A = (Q, \Sigma, \Delta, q_0, F)$, where

- Q is a set of states;
- Δ is a set of transitions (or edges);
- q_0 is the start state;
- F is a set of states from Q which are final states.

Symbols $\delta.to$ and $\delta.from$ refer to the source and destination states of an edge δ, respectively. An edge also carries a symbol taken from the alphabet Σ. This edge symbol, denoted by $\delta.sym$ can also be the empty symbol ϵ, in which case a state transition is instantaneous – when matching against an input pattern, no input symbol needs to be processed for an ϵ transition to take place. A pattern is a sequence of symbols from Σ. For a non-ϵ edge, a given transition takes place when the input symbol is identical to the edge's symbol. When a final state is active, the pattern has been matched.

A regular expression is converted to an equivalent automaton in a recursive manner [11]. First, terminal automata are built for the symbols of the regular expression. Next, these automata are recursively combined according to the regular expression operators comprising the given expression. Choice and concatenation of two sub-automata involves combining the sub-automata using ϵ edges. The Kleene closure of an automaton is created by adding ϵ edges for bypassing the automaton (empty matching), and re-triggering the automaton (multiple consecutive matching). The construction procedure produces a Nondeterministic Finite Automaton containing ϵ transitions (NFA). An automaton can be determinized, hence producing a deterministic finite automaton (DFA). Procedures for removing ϵ transitions, minimizing and determinizing classical automata are well known [11].

Definition 1: An automaton for which no ϵ edge is present and no state has more than one outgoing edge with the same symbol is a *Deterministic Finite Automaton (DFA)*, otherwise it is a *Nondeterministic Finite Automaton (NFA)*.

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

The *Boolean Layer* in PSL is built around the Boolean expressions of the underlying HDL. 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. *Sequences*, or Sequential-Extended Regular Expressions (SEREs), are used to specify temporal chains of events of Boolean primitives. SEREs are defined as follows [12]. If b is a Boolean expression and r, r_1 and r_2 are SEREs, the following expressions are SEREs:

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

Here, the curly brackets are the equivalent of the parentheses in conventional regular expressions, and the semicolon represents temporal 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 is the Kleene star, and the $|$ operator corresponds to sequence disjunction (choice).

The colon operator denotes *sequence fusion*, which is a concatenation in which the last Boolean expression occurring in the first SERE must intersect (i.e. both held to be true) with the first Boolean primitive occurring in the second SERE. Empty sequences in either side do not result in a match. The *length matching sequence intersection* operator (&&) requires that both argument sequences occur, and that both sequences start and terminate at the same time. The $[*0]$ operator is the empty SERE and is equivalent to the ϵ expression mentioned previously. The ϵ can be seen as a primitive which spans no clock cycles. All 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 PSL SERE sugaring operators are shown next, which we arrange to use as rewrite rules in preparation for the next section. Some rules are presented differently than in [12]; we believe the rules shown below offer a more intuitive form. b is a Boolean expression; r is a SERE; l, h and c are nonnegative integers with $h \geq l$; and the $\overset{\rightarrow}{=}$ symbol indicates equivalency, with a preferred direction to be used as a rewrite rule. $^+$ also denotes a positive integer.

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

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 $r =$ "true" is implied. The $[=]$ 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, sequences can appear in two different semantic contexts.

Definition 2: *Conditional mode*. Sub-statement semantic context for which the detection of a sequence must be performed. For each start condition, the result signal is triggered each and every time the chain of events described by the sequence is observed.

Definition 3: *Obligation mode*. Semantic context for which the failure of a sequence must be identified. For each start condition, if the chain of events described by the sequence does not occur, the result signal is triggered (for a given start condition, the first and only the first failure is identified).

For example, in "always $(\{b_1\,;b_2\}\ |\text{–>}\ \{b_3\,;b_4\})$", the left sequence is in *conditional* mode because its occurrence is used to trigger a condition. On the other hand, the right sequence is in *obligation* mode because its failure to occur indicates an error. In our semantics for dynamic verification, obligation mode is not a proper negation of conditional mode.

III. Transforming Sequences into Circuits

A. Sequence Automata

Given that the symbol encoding we use in the automata for SEREs is different than in classical automata, we introduce *Sequence Automata* (s-automata), denoted by $\mathcal{A}$. The encoding differs from conventional regular expressions because in SEREs, the alphabet Σ represents arbitrary Boolean expressions. Our encoding of Boolean primitives to s-automaton symbols is to assign a distinct symbol s_i to each Boolean primitive b_i, without concern for the fact that two Boolean primitives may simultaneously evaluate to true. This introduces a fundamental difference with normal REs: in regular expressions, *one* symbol is received at each step, while the Boolean expression symbol encoding causes multiple symbols to be received at each step. A similar encoding was also used in the co-universal automata for PSL model checking in [13].

It is possible to create a symbol encoding which represents the power set of the Boolean primitives, such that one and only one symbol is received at each step during the matching. However, when a SERE references many Boolean primitives, the exponential increase of symbols and edges quickly becomes impractical.

```
1:  FUNCTION: INTERSECT_S-AUTOMATA($\mathcal{A}_1$, $\mathcal{A}_2$)
2:  **create** new s-automaton $\mathcal{A}$
3:  **push** state $(0,0)$ onto construction stack
4:  **while** stack is non-empty **do**
5:      $(i,j) \leftarrow$ **pop** stack
6:      **create** the state labeled $(i,j)$ in $\mathcal{A}$
7:      **for** each edge $\alpha$ of state $i$ in $\mathcal{A}_1$ **do**
8:          **for** each edge $\beta$ of state $j$ in $\mathcal{A}_2$ **do**
9:              **add** a new edge $\delta$ to state $(i,j)$
10:             $\delta.sym \leftarrow \alpha.sym \wedge \beta.sym$
11:             $\delta.to \leftarrow (\alpha.to, \beta.to)$
12:             **push** $(\alpha.to, \beta.to)$ un-redundantly onto stack
13:     **mark** state $(i,j)$ final iff $i$ and $j$ are final states
14: **return** $\mathcal{A}$
```

Fig. 1. Sequence-automata intersection algorithm.

B. Building Sequence Detection Automata

Sequence automata are constructed recursively, during the traversal of a sequence expression. Numerous syntactical sugaring operators are dynamically rewritten as they are encountered such that they do not appear as distinct operators.

1) Conventional Regular Expression Operators: The conventional operators for regular expressions are concatenation, choice and the Kleene star, as presented in Section II. The Boolean expression encoding scheme produces nondeterministic automata because from a given state, two distinct symbols can cause simultaneous outgoing transitions when their respective Boolean expressions are both true. The NFA construction for conventional operators can be reused for sequence automata given that s-automata are also nondeterministic.

2) Length-Matching Intersection: Typical automata intersection [11], which equates to building a *product automaton* of both arguments, is incompatible with the Boolean expression symbol encoding. If the traditional intersection procedure is applied to two automata which have no symbol in common (i.e. no syntactically equal symbols), an empty automaton results. This automaton can obviously not detect the intersection of two sequences which use disjoint sets of primary symbols.

To implement intersection for sequence-automata, the condition on syntactic equality of symbols must be relaxed and the conjunction of symbols must be considered by the algorithm. To consider all relevant pairs of edges, the intersection automaton is built using the subset construction technique, as is typically done in the determinization algorithm [11]. This technique, characterized by its creation stack, is visible in our intersection algorithm in Fig.1. The state-creation stack is the core of the algorithm (line 3), and is initialized to the pair of initial states of both argument automata (line 2). A state in the intersection automaton is labeled by a state pair (i,j), where i and j correspond to states from the first and second argument automata respectively. The two **for** loops intersect two states, and the **while** loop together with the stack create the entire intersection automaton. A new state (i,j) is a final state if

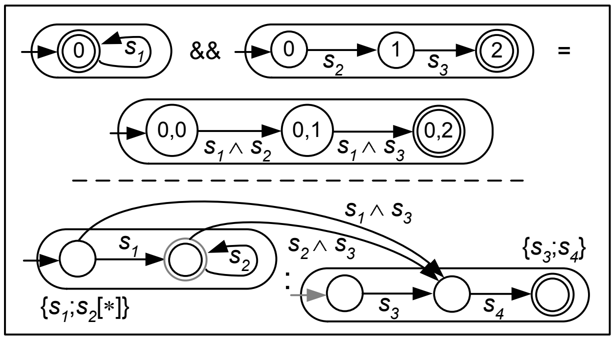

Fig. 2. Examples: intersection (top) and fusion (bottom).

```
1: FUNCTION: FUSE_S-AUTOMATA($\mathcal{A}_L$, $\mathcal{A}_R$)
2: create s-automaton $\mathcal{A}$ with $\mathcal{A}_L$ and $\mathcal{A}_R$ (both are disjoint)
3: for each edge $\delta_L$ in $\mathcal{A}_L$ that hits a final state do
4:     for each edge $\delta_R$ in $\mathcal{A}_R$ that leaves the start state do
5:         create new edge $\delta$ from $\delta_L.from$ to $\delta_R.to$
6:         $\delta.sym \leftarrow \delta_L.sym \wedge \delta_R.sym$
7: mark final states in $\mathcal{A}_L$ as a non-final states
8: set the start state to $\mathcal{A}_L$'s start state
9: return $\mathcal{A}$
```

Fig. 3. Sequence-automata fusion algorithm.

and only if states i and j are final states in their respective automata. Because of the subset construction approach, new states and edges are only created for the reachable states of the resulting automaton.

An example depicting intersection is shown in the top part of Fig.2 for $\{s_1[*]\}$ && $\{s_2; s_3\}$. Sequence intersection produces a sequence-automaton that has in the worst case mn states, where m and n are the sizes of the input automata. The algorithm is easily proven to terminate.

3) Fusion: As observed in Section II, sequence fusion can be seen as an overlapped concatenation. Our algorithm for performing the fusion of two sequence-automata is shown in Fig.3. The algorithm starts by incorporating both argument automata into a new automaton, such that they are disjoint (line 2). From there, fusion edges are created from edges that hit final states in the left automaton and edges that leave the start state of the right automaton. This algorithm has similarities to the independently developed algorithm in [9].

Sequence fusion produces a sequence-automaton that has $m + n$ states, where m and n are the sizes of the input automata. When the start state of the right side automaton is non-final (ex: Fig.2), this state becomes useless and is removed during minimization. Because empty strings on either side do not result in a match, marking all states in the left automaton as non-final states is correct. The number of edges added can be easily determined by examining the two **for** loops in the algorithm. An illustration of sequence fusion is shown in the bottom half of Fig.2 for $\{s_1; s_2[*]\} : \{s_3; s_4\}$.

C. Building Sequence-Failure Automata

The sequence-automata built by the algorithms from the previous subsection perform precisely the task described by Definition 2, with respect to conditional mode sequences. In obligation mode sequences however, each start condition indicates that the sequence should occur, and that its non-fulfillment indicates an error. Alternatively, this can also be seen as the detection of the first failure. Therefore, a separate procedure is required to transform the sequence detection automaton into a first-failure detection automaton, in accordance with Definition 3 (obligation mode sequences).

```
1: FUNCTION: FIRSTFAIL_S-AUTOMATON($\mathcal{A}$)
2: $\mathcal{A} \leftarrow$ S-Determinize($\mathcal{A}$)
3: add a fail state $f$ to $\mathcal{A}$
4: for each state $s$ in $\mathcal{A}$ do
5:     for each Boolean assignment of $b_i$ in edges from $s$ not
       handled by an edge do
6:         create new edge $\delta$ from state $s$ to state $f$
7:         $\delta.sym \leftarrow$ symbol for Boolean assignment above
8:     remove edges from state $s$ which lead to a final state
9: mark state $f$ as the final state
10: return $\mathcal{A}$
```

Fig. 4. Failure transformation algorithm.

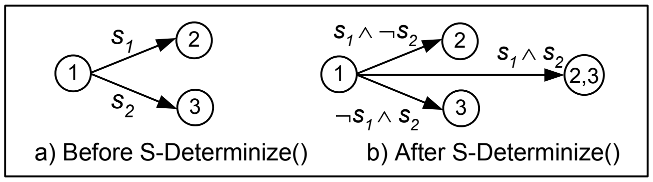

Fig. 5. Determinizing sequence-automata.

Fig.4 shows the transformation algorithm used to produce a failure detection automaton from a detection automaton. Under the Boolean encoding of symbols, Definition 1 for DFAs is no longer sufficient. Classical determinization "sees" only symbols, and does not consider the possibility that these symbols represent Boolean expressions, many of which can simultaneously be true. Our determinization algorithm (S-Determinize(), line 2) is based on the classical determinization algorithm [11], with added features that take into consideration the Boolean nature of symbols (as is also done in [13]). The S-Determinize() algorithm produces *deterministic sequence-automata*. The effect of this algorithm is shown using a small example in Fig.5. The left side shows a nondeterministic s-automaton and the right side shows the result of applying S-Determinize() ("2,3" is a label for a new state). In the determinized s-automaton, no conditions will allow state 1 to transition into more than one state. If the automaton in the left side of Fig.5 was not a sequence automaton, it would already be deterministic.

Determinization produces an automaton with at most 2^m states, where m is the number of states of the input automaton; however in practice, the resulting automaton is often much smaller. The failure transformation algorithm is therefore also exponential (worst case) in the number of input states. The main idea behind this algorithm is to catch the case where the automaton "dies" (i.e. from the active state, no transitions are taken). In order for the automaton to be in one and only one state, for a given start condition, it must be deterministic. The **for** loop at line 5 adds precisely the Boolean conditions that lead to the "death" of the automaton, hence the failure automaton is produced. Line 8 produces the necessary pruning, given that we are only interested in the first failure. The determinization also explains why multiple succeeding start conditions can be processed in a pipelined (or threaded) manner, by the failure automaton. This allows failures to be identified in a continual and dynamic manner during execution. The failure algorithm actually implements a form of negation; however this does not correspond to classical automata negation for many reasons, most notable of which is the dynamic semantics imposed by Definition 3 (obligation mode).

IV. Experimental Results

Hardware implementations are well suited for nondeterministic automata [8]; in our case, sequence-automata are converted to hardware in a similar manner, using flip-flops and combinational logic. In this section, the assertion circuits produced by MBAC and FoCs are compared, using examples that showcase both obligation and conditional sequences. Temporally simple assertions, such as those used for verifying most bus protocols are not informative for evaluating a checker generator, as they span very few clock cycles. However, as assertions become more popular and verification engineers become more adept at writing assertions, checker generators must be able to follow.

The comparison is partitioned into two categories: hardware and software metrics. Software metrics are obtained using the Modelsim simulator (version 6.1f SE). This simulator implements PSL and is used as a golden reference to ensure proper behavior of the assertion circuits. 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 emulation. Because speed may also be an issue, the maximum operating frequency for the worst clk-to-clk path is reported.

Ideally, the assertion circuits that are produced should be small, fast and should provide the correct behavior. In order for the software metrics to be meaningful, the assertions are made to trigger often during a simulation run. To accomplish this, primary signals supplied by the testbench are pseudo-randomly generated with different probabilities. When comparing with FoCs, MBAC's final result signals are also sampled by a FF. The random stimulus comparison is obviously not a proof that the circuits generated by MBAC are correct, however it does offer reasonable assurance.

The *assertion distance* is introduced to compare the behavior of two assertion circuits' outputs. For two given traces of assertion signals, the assertion distance is defined as the number of clock cycles in which the two assertion signals disagree. The signals in question are typically from two different implementations of the *same* PSL assertion.

A. Conditional-Mode Sequences

In this subsection, we empirically evaluate the efficiency and behavior of the intersection and fusion operators in conditional mode sequences. Regarding the semantics of PSL in dynamic verification, it should be noted that some assertions interpreted by simulators such as Modelsim will only trigger once for any given start condition. Between MBAC's assertion circuits and Modelsim, the distance is not incremented when an assertion circuit output triggers *and* Modelsim's assertion does not. When such a condition occurs, a residual distance is instead incremented. Residual distances are an indication that MBAC is reporting more failures, which can then be exploited for debugging purposes. Residual distances are noted "r n". When applicable, the residual distance is well anticipated because of the multiple paths in the corresponding sequence. In all cases, 10^5 pseudorandom test vectors are supplied by the testbench.

The FoCs and MBAC checker generators are evaluated with the set of assertions shown in Table I. In the comparison table, N.A. means Not Applicable, and appears when an assertion circuit contains only one FF and the FF has no feedback path (the MHz is a clk-to-clk figure). In all cases, the circuits produced by MBAC are smaller and have an equal or higher operating frequency; their behavior is also correct and better suited for debugging purposes, as more assertion violations get reported.

B. Obligation-Mode Sequences

In this subsection both tools are evaluated using sequences that appear in obligation mode. The FoCs and MBAC checker generators are evaluated with the set of assertions shown in Table II. The only case of a non-zero distance to Modelsim occurs because MBAC's circuits are able to identify certain failures *earlier* than Modelsim. This arises because MBAC is sometimes able to reach a terminal state earlier when evaluating a given sequence. Contrary to the conditional mode results, the circuits generated by FoCs are similar to MBAC's. However, the strength of our approach becomes apparent when increasing the complexity of sequences. In all of our experiments, checkers were generated instantly by our tool.

V. Conclusion and Continuing Work

As assertion-based verification and emulation become increasingly important in verification, a practical tool for generating efficient hardware assertion checkers is a must. We have shown how to generate efficient circuits from sequences and introduced algorithms for implementing the sequence fusion and intersection operators. Such operators are not typically used in conventional regular expressions. The sequences used in assertion languages are not exclusively used for pattern matching (conditional mode); sequences also appear in obligation mode. For the latter, we have introduced an algorithm for creating a failure detection automaton from a classical matching automaton. At the core of these algorithms is the efficient symbol encoding of our automata-based methods. Experimental results show important improvements in terms of the resource usage, behavior and capability of assertion

TABLE I

BENCHMARKING OF CONDITIONAL-MODE SEQUENCES (N.A. = NOT APPLICABLE).

	Hardware Emulation Metrics						**Asr. Distances**	
Assertions		MBAC			FoCs		MBAC–	MBAC–
“assert never Sx;”, where Sx is	FF	LUT	MHz	FF	LUT	MHz	FoCs	MSim
S1: { a;d;{b;a}[*2:4];c;d }	**12**	**12**	622	25	24	622	0	0 r1910
S2: { {a[*0:1];b[*0:2]} : {c[*0:1];d} }	**2**	**3**	**680**	33	30	326	0	0 r2366
S3: { {e;e} within {c;d;a;b;c} }	**11**	**11**	**622**	30	29	521	9507	0
S4: { {{b;c[*1:2];d}[+]} && {b;{e[−>2:3]};d} }	**16**	**20**	422	40	48	422	3727	0 r31
S5: { {a[*];b[*1:3]} | {c;d[*1:2];e} }	**4**	**4**	**622**	24	23	454	0	0 r445
S6: { {a|b} ; {{c[*]} && {d[*1:3]}} : {e} }	**4**	**5**	**622**	14	13	487	0	0 r300
S7: { {a|b} ; {{c[*]} && {d[=4]} } : {e}}	**6**	**9**	**616**	22	24	487	0	0 r4490
S8: { {a|b} ; {{c[*]} && {d[−>1]}} }	**2**	**4**	622	5	6	622	0	0
S9: { {[*];a} && {b[=0]} }	**1**	**2**	**N.A.**	6	4	622	0	0
S10: { a ; {b;c;d} & {e;b;a;d} ; a }	**6**	**6**	**680**	13	12	622	0	0
S11: { {a;b[*1:3]} & {c[*2:4]} }	**6**	**9**	**479**	91	117	285	37396	0 r5637
S12: { {a[*]} : {b[*]} }	**1**	**2**	**680**	7	7	483	0	0 r1867
S13: { {a;[*];b} && {c[*1:5];d} }	**6**	**8**	**487**	18	22	425	4166	0 r609
S14: { {c[−>1]} && {d[=0]} }	**1**	**2**	**N.A.**	5	4	622	0	0
S15: { {a| b};{{c[*]} && {d[*1:3]}}:{e} }	**4**	**5**	**622**	14	13	487	0	0 r1800
S16: { {a| b};{{c[*]} && {d[*1:6]}}:{e} }	**7**	**9**	**487**	20	21	483	0	0 r1858

TABLE II

BENCHMARKING OF OBLIGATION-MODE SEQUENCES (N.O. = NO OUTPUT).

	Hardware Emulation Metrics						**Asr. Distances**	
Assertions		MBAC			FoCs		MBAC–	MBAC–
“assert always {a} |=> Sx;”, where Sx is	FF	LUT	MHz	FF	LUT	MHz	FoCs	MSim
S1: { b;c[*];d }	3	5	622	3	**4**	622	3272	0
S2: { {b;c;d} & {e;d;b} }	**4**	**6**	**483**	N.O.			N.O.	0
S3: { e;d;{b;e}[*2:4];c;d }	**15**	**21**	**378**	N.O.			N.O.	0
S4: { b ; {c[*0:4]} & {d} ; e }	7	11	**487**	7	**10**	359	3973	0
S5: { b ; {c[*0:6]} & {d} ; e }	**9**	**15**	**428**	N.O.			N.O.	0
S6: { {{c;d}[*]} && {e[−>4]} }	**9**	**13**	**420**	10	15	357	0	0
S7: { {{c;d}[*]} && {e[−>6]} }	**13**	**19**	**420**	N.O.			N.O.	0
S8: { {{c[*1];d}[+]} && {e[−>2]} }	**5**	**7**	**517**	6	10	425	0	0
S9: { {{c[*1:2]}[+]} && {e[−>2]} }	**3**	**4**	**616**	N.O.			N.O.	0
S10: { {{c[*1:2];d}[+]} && {e[−>2]} }	**9**	**20**	**355**	N.O.			N.O.	0
S11: { {{c[*1:3];d}[+]} && {{e[−>2:3]};d} }	**20**	**44**	**305**	N.O.			N.O.	209
S12: { {{b;c[*1:2];d}[+]} : {{e[−>]};d} }	**22**	**58**	**317**	N.O.			N.O.	0
S13: { {{b;c[*1:2];d}[+]} : {b;{e[−>2:3]};d} }	**65**	**192**	**255**	N.O.			N.O.	0
S14: { b ; {{c[*0:2]} ; {d[*0:2]}}[*] ; e }	**3**	**5**	**622**	7	15	425	26	0 r2061
S15: { {{c[*1:2];d}[+]} & {e[−>2]} }	**4**	**5**	**561**	N.O.			N.O.	0

circuits. Using our tool, assertions can be used in hardware emulation, in silicon debug tools, as well as in efficient simulations that do not support assertion languages such as PSL.

REFERENCES

[1] 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.

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

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

[4] M. Oliveira and A. Hu, “High-Level Specification and Automatic Generation of IP Interface Monitors,” *39th Design Automation Conference, ACM Press*, pp. 129–134, 2002.

[5] S. Edwards, “An Esterel Compiler for Large Control–Dominated Systems,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 21*, 2002.

[6] A. Seawright and F. Brewer, “High–Level Symbolic Construction Techniques for High Performance Sequential Synthesis,” *30th Design Automation Conference, ACM Press*, pp. 424–428, 1993.

[7] P. Raymond, “Recognizing Regular Expressions by Means of Dataflow Networks,” *23rd International Colloquium on Automata, Languages and Programming*, pp. 336–347, 1996.

[8] R. Sidhu and V. Prasanna, “Fast Regular Expression Matching using FPGAs,” *9th Annual IEEE Symposium on Field–Programmable Custom Computing Machines (FCCM’01)*, pp. 227–238, 2001.

[9] 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.

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

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

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

[13] S. Ruah, D. Fisman, and S. Ben-David, “Automata Construction for On-The-Fly Model Checking PSL Safety Simple Subset,” IBM, Tech. Rep. H-0234, 2005.