Abstract—Synthesis techniques take realizable Linear Temporal Logic specifications and produce correct circuits that implement the specifications. The generated circuits can be used directly, or as miters that check the correctness of a logic design. Typically, those techniques generate non-deterministic finite state automata, which can be determined at a possibly exponential cost. Recent results show multiple advantages of using deterministic automata in symbolic and bounded model checking of LTL safety properties. In this paper, we present a technique with a supporting tool that takes a sequential extended regular expression specification $\Phi$, and a logic design implementation $S$, and generates a sequential circuit $C$, expressed as an And-Inverted-Graph, that checks whether $S$ satisfies $\Phi$. The technique passes the generated circuit $C$ to ABC, a bounded model checker, to validate correctness.

We use free input variables to encode the non-determinism in $\Phi$ and we obtain a number of states in miter linear in the size of $\Phi$. Our technique succeeds to generate the input to the model checker while other techniques fail because of the exponential blowup, and in most cases, ABC succeeds to either find defects in the design that was otherwise uncheckable, or validate the design. We evaluated our technique against several industrial benchmarks including the IBM arbiter, a load balancer, and a traffic light system, and compared our results with the NuSMV framework. Our method found defects and validated systems NuSMV could not validate.

I. INTRODUCTION

Safety critical systems such as medical and navigation control devices rely on digital systems in order to provide accurate services. Verification techniques, such as symbolic and bounded model checking, address the correctness of digital systems with respect to formal specifications written in languages such as linear temporal logic (LTL). Sequential extended regular expressions (SERE) form a subset of the Property Specification Language (PSL) that constitute a practical way to specify logic designs [1]. SERE covers a practical subset of LTL.

Automated synthesis tools such as Wring [2], Lily [3], and UNBEAST [4] take an LTL specification and generate a correct implementation. Validation tools such as Focs [5], NuSMV [6], and SPIN [7] take a specification and an implementation and check whether the implementation satisfies the specification. They either provide a proof of correctness, a counterexample, or an inconclusive result when they reach their computational boundaries [8].

NuSMV [6] and COSPAN [9] typically translate the design $S$ and the negation of the LTL specification $\Phi$ into non-deterministic finite state automata (NDFA) $M_S$ and $M_{\neg \Phi}$ (typically using Büchi automata), respectively, and then perform symbolic model checking on the resulting cross product automaton [10], [11]. This results in an online determinization of the assertion automaton and thus the state space explosion problem is inherent to symbolic model checking [10].

The majority of the LTL properties to be verified are safety properties, to which finite violating counterexamples can be found. Therefore, researchers consider translating the LTL specifications into deterministic finite state automaton (DFA) $M_S$ and $M_{\Phi}$ (typically using Büchi automata), respectively, and then perform symbolic model checking on the resulting cross product automaton [10], [11]. This often limits the ability of synthesis tools to generate input to the model checkers for verification. NuSMV uses several abstraction and reduction techniques, such as the cone of influence reduction [14] and other Binary Decision Diagrams (BDD) based techniques [15], in order to avoid such a problem.

In this paper, we present a technique and a supporting tool that takes an SERE specification $\Phi$ and an implementation of it $S$, and generates a sequential circuit $C$ that checks whether $S$ satisfies $\Phi$. Our technique encodes the non-determinism in $\Phi$ using additional free variables, and generates an equisatisfiable sequential circuit $C_\Phi$ such that $C_\Phi$ has a number of states linear in the size of $\Phi$. Informally, a sequential circuit $C$ with a designated output $o$ therein is equisatisfiable to an SERE specification $\Phi$ when $o$ is satisfiable if and only if $\Phi$ is satisfiable. The circuit $C_\Phi$ can not be used as an implementation of $S$ and can only be used as a miter in
model checking tools to validate an implementation of \( \Phi \). The technique translates the implementation \( S \) into a sequential circuit \( C_S \) and builds \( C \) as the composition of \( C_\Phi \) and \( C_S \). The technique then applies the ABC model checker on the generated sequential circuit \( C \) and checks for correctness.

Encoding non-determinism using free variables is a textbook technique \[13\]. Reportedly, it might have been used in existing tools such as “smvtoaig” for a limited subset of the “smv” designs. Up to our knowledge, we are the first to use this technique in an open source tool to enable the verification of logic design against SERE specifications. Our technique enables the ABC model checker to find defects and prove the correctness of systems where it is not possible with existing techniques.

We implemented and evaluated our techniques with benchmarks from UNBEAST \[4\] and LILY \[1\], in addition to the IBM arbiter presented in \[16\]. We provide our tool, the appendices of the paper including proofs, and the benchmarks for the experiments online \[1\]. Our technique was able to find problems in several designs where it was not possible before. The supporting tool allows the user to:

- **prove** that an implementation satisfies an SERE property using satisfiability and bounded model checking,
- **debug** the implementation and the specification using the generated counterexample, and
- **simulate** the implementation and the specification and inspect the results.

The rest of this paper is organized as follows. Section II presents some preliminary information, Section III motivates our approach using a simple example. The core of the synthesis technique is presented in section IV. We describe our implementation in Section V and show a summary of the experimental results in Section VI. Related work is summarized in Section VII and we conclude in Section VIII.

II. PRELIMINARIES

Let \( \mathbb{A} \) be a set of atomic propositions. The mapping \( \mathbb{A} \to \mathbb{B} \) denotes a valuation to the atomic propositions in \( \mathbb{A} \) where \( \mathbb{B} = \{ \text{true}, \text{false} \} \). Let \( \mathbb{V} = (\mathbb{A} \to \mathbb{B}) \) be the set of all such valuations.

SERE formulae range over the alphabet \( \Sigma = \mathbb{A} \cup \{ ; \times, \land, \lor, \neg, (, )\} \), where (1) \( \land, \lor \) are Boolean binary operators denoting conjunction and disjunction, respectively, (2) \( ; \) is a sequential binary operator denoting temporal next, (3) \( \neg \) is a Boolean unary operator denoting logical negation, and (4) \( * \) is a sequential unary operator denoting zero or more times.

**Definition (SERE terms).** An atomic proposition in \( \mathbb{A} \) is an SERE term. If \( t_1 \) and \( t_2 \) are SERE terms, then \( t_1 \land t_2 \), \( t_1 \lor t_2 \), \( (t_1) \) and \( \neg t_1 \) are SERE terms. We denote by \( \mathbb{T} \) the set of all SERE terms.

**Definition (SERE formula).** An SERE term is an SERE formula. Given \( \phi \) and \( \psi \) are SERE formulae and \( t \) is an SERE term, then \( t \ast \), \( \phi \land \psi \), \( \phi \lor \psi \), and \( \phi \lor \psi \) are all SERE formulae. We denote by \( \mathbb{S} \) the set of all SERE formulae.

A valuation \( v \in \mathbb{V} \) satisfies an atomic proposition \( a \in \mathbb{A}, v \models a \) iff \( v \) maps \( a \) to true; we denote that also by \( v(a) = true \). A trace \( \rho = \langle v_1, v_2, \ldots, v_i, \ldots, v_n \rangle, i \in [1 \ldots n] \) is a sequence of valuations. We denote by (1) \( \rho = \rho_1 \circ \rho_2 \) the concatenation of the traces \( \rho_1 \) and \( \rho_2 \), and (2) \( \rho(t, i) \) the value of term \( t \) at the \( i \)th entry of \( \rho \).

**Definition (SERE term semantics).** Let \( \rho \) be a trace, and let \( e_1 \) and \( e_2 \) be SERE terms.

- If \( e_1 \in \mathbb{A} \) then \( \rho \models e_1 \) iff \( | \rho | = 1 \) and \( \rho(e_1, 1) \)
- \( \rho \models \neg e_1 \) iff \( | \rho | = 1 \) and \( \rho \neq e_1 \)
- \( \rho \models e_1 \land e_2 \) iff \( \rho \models e_1 \) and \( \rho \models e_2 \)
- \( \rho \models e_1 \lor e_2 \) iff \( \rho \models e_1 \) or \( \rho \models e_2 \)

We denote by \( [e] \) all the valuations that satisfy term \( e \).

Let \( \psi \) be an SERE formula of the form \( x_1 y_1; x_2 y_2; \ldots; x_n y_n \), where \( i \in [1 \ldots n] \) \( x_i \in \mathbb{T} \), \( y_i \in \{ \epsilon, * \} \), and \( \epsilon \) is the empty string.

**Definition (SERE formula semantics).** Let \( \rho \) be a trace and \( \psi \) be an SERE formula. We say \( \rho \) satisfies \( \psi(\models \psi) \) in the following cases.

- \( \psi = x_1 y_1, y_1 = \epsilon \) iff \( | \rho | = 1 \) and \( \rho(x_1, 1) \)
- \( \psi = x_1 y_1, y_1 = * \) iff
  - \( \rho = \epsilon \),
  - \( | \rho | = 1 \) and \( \rho \models x_1 \), or
  - there exists traces \( \rho_1, \rho_2 \) such that \( \rho_1 \neq \epsilon \), \( \rho = \rho_1 \circ \rho_2 \), \( \rho_1 \models \psi \), and \( \rho_2 \models \psi \)
- \( \psi = x_1 y_1; x_2 y_2 \) iff there exists traces \( \rho_1, \rho_2 \) such that \( \rho = \rho_1 \circ \rho_2 \), \( \rho_1 \models x_1 y_1 \), and \( \rho_2 \models x_2 y_2 \)
- \( \psi = \phi_1; \phi_2 \) where \( \phi_1, \phi_2 \) are formulae iff there exists traces \( \rho_{\text{prefix}}, \rho_1, \rho_2, \rho_{\text{suffix}} \) such that \( \rho = \rho_{\text{prefix}} \circ \rho_1 \circ \rho_2 \circ \rho_{\text{suffix}} \), \( \rho_{\text{prefix}} \models \phi_1 \), and \( \rho_2 \models \phi_2 \)
- \( \psi = \phi_1 \land \phi_2 \) iff \( \rho \models \phi_1 \) and \( \rho \models \phi_2 \)
- \( \psi = \phi_1 \lor \phi_2 \) iff \( \rho \models \phi_1 \) or \( \rho \models \phi_2 \)

We denote by \( [\psi] \) all the traces that satisfy formula \( \psi \).
Definition (Deterministic finite state automata). A deterministic finite state automata (DFA) is a tuple \( M = (Q, I, F, \Sigma, L, \delta) \) where \( Q = \{s_0, s_1, \ldots, s_n\} \) is the set of states of \( M \), \( I \subseteq Q \) is the set of initial states, \( F \subseteq Q \) is the set of accept states, \( \Sigma = \mathcal{V} \) is the input alphabet of \( M \), \( L \subseteq \mathbb{T} \) is a set of transition labels such that \( \delta = Q \times L \rightarrow Q \) is the state transition function. Note that labels with joint alphabet symbols (e.g., \( a \), \( a \land b \), \( a \lor b \)) are not allowed on edges outgoing from a state \( s \) in order to keep the transitions deterministic.

The semantics of DFA are defined in the typical manner. A sequence of input valuations \( \rho = \langle v_0, v_1, \ldots, v_{n-1} \rangle \), determines a sequence of state transitions \( \sigma = \langle s_0, s_1, \ldots, s_n \rangle \), \( s_0 \in I \), and \( s_{i+1} = \delta(s_i, e) \) where \( e \in L \) and \( v_i \in [e] \). We say \( \rho \) satisfies \( M (\rho \models M) \) iff \( s_n \) is an accept state of \( M \); \( (s \in F) \).

Definition (Equisatisfiability). We say a DFA \( M \) is equisatisfiable to an SERE formula \( \psi \) iff \( M \) is satisfiable iff \( \psi \) is satisfiable. That is \( \exists \rho. \rho \models M \iff \exists \rho'. \rho' \models \psi. \)

III. MOTIVATING EXAMPLE

Consider the SERE formula \( \psi = a; b; c \). NDFA \( M \) in Figure 1 simulates \( \psi \) with non-deterministic transitions in its initial state. Once \( M \) receives a valuation where \( a \) is \( \text{true} \), it can move into state \( s_1 \) or remain in \( s_0 \) since \( \delta(s_0, a) = \{s_0, s_1\} \).

Typically, an NDFA \( M \) is translated into a DFA \( M' \) using subset construction with a possible exponential blowout in the number of states \([13]\). In brief, states in \( M' \) are subsets of states in \( M \) and transitions are constructed to make \( M' \) equivalent to \( M \), yet deterministic. Figure 2 shows a DFA equivalent to \( M \) produced using the JFLAP tool \([17]\).

Instead, we encode the non-determinism using an additional free atomic proposition \( r \) as shown in Figure 3. This results in DFA \( M^a \) that is equisatisfiable to \( \psi \) and that has a number of states linear in the number of terms in \( \psi \). We use \( M^a \) with symbolic and bounded model checkers wherever it is expensive or impossible to generate an equivalent DFA for \( \psi \). Our technique leaves it to the model checker to efficiently handle the free variables added by our synthesis technique. In practice, even though our technique does not reduce the inherent complexity of the problem, it enables the application of several reduction and abstraction transformations available in model checkers such as ABC to reduce and solve the problem. These are not applicable without our technique.

Notice that, for each trace \( \rho \) that satisfies \( a; b; c \), there is a trace of \( r \) values that makes \( \rho \) satisfiable for the DFA in Figure 3. In particular, set \( r \) to \( \text{true} \) where the matching sequence starts in \( \rho \) and to \( \text{false} \) otherwise.

IV. EQUISATISFIABLE DFA

Given an SERE formula \( \psi \), we want to efficiently construct a DFA \( M \) with a number of states linear in the size of \( \psi \) that is equisatisfiable to \( \psi \) such that the trace \( \rho \) that satisfies \( M \) also satisfies \( \psi \). We focus on the two sources of non-determinism: the initial states and the \( * \) operator.

We first consider formulae \( \psi \) of the form \( \psi = x_1 y_1; x_2 y_2; \ldots; x_n y_n \) where \( \mathcal{A} \) and \( \mathbb{T} \) denote the atomic propositions and SERE terms of \( \psi \), respectively, \( x_i \in \mathbb{T} \), \( y_i \in \{\epsilon, *\} \), and \( \epsilon \) is the empty string. We want to construct a DFA \( M = (Q, I, F, \Sigma', \mathbb{T}', \delta) \) where \( Q = \{s_0, s_1, \ldots, s_n\} \), \( I = \{s_0\} \), and where each state \( s_i \) corresponds to a term \( x_i \) in \( \psi \). The other components \( F, \Sigma', \mathbb{T}' \), and \( \delta \) will be discussed later.

Consider the initial state \( s_0 \), and consider an input valuation \( v \) that matches \( x_1 \) the first term in \( \psi \). The DFA \( M \) needs to allow for two possibilities: (1) \( v \) is part of the sequence matching the terms of \( \psi \), and
(2) $v$ is ignored and next inputs are considered as the match to the first term in $ψ$. For example, consider $ψ = a; b$ where $A = \{a, b\}$ and consider the trace $ρ = \langle v_1, v_2, v_3, v_4 \rangle$ where $v_1 = \{(a, true)\}, v_2 = \{(a, true)\}, v_3 = \{(a, true)\}$, and $v_4 = \{(a, false)\}$. The subtrace $\langle v_2, v_3 \rangle$ of $ρ$ matches $ψ$ while $v_1$ matches only the first term $a$ in $ψ$. Also $\langle v_3, v_4 \rangle$ matches $ψ$. The DFA $M$ should allow a choice of whether to stay in the initial state $s_0$ or to start the acceptance chain of transitions.

Consider the subformula $x_1; x_2$ which specifies that input valuations that match the term $x_1$ occur zero or more times in succession followed by a valuation that matches the term $x_2$. By definition, this includes non-determinism at every step. Once the valuation that matches $x_1$ is presented, $M$ should allow for more valuations matching $x_1$, and since we are restricting $s_1$ to correspond to $x_1$, $M$ stays at the same state. $M$ should as well allow for valuations matching $x_2$ by transitioning to state $s_2$.

Consider the SERE formula $ψ = a; b*; a$ where $A = \{a, b, c\}$. Consider the trace $ρ = \langle v_1, v_2, v_3, v_4, v_5 \rangle$ where $v_1 = \{(a, true)\}, v_2 = \{(a, true)\}, v_3 = \{(a, true)\}$, $v_4 = \{(a, false)\}$, and $v_5 = \{(a, true)\}$. Again, $ρ$ can satisfy $ψ$ in several ways. One way is to consider accepting the subtrace $\langle v_1, v_2, v_3 \rangle$, and another is consider accepting the subtrace $\langle v_3, v_4, v_5 \rangle$. Once an input valuation such as $v_3$ that matches the second $a$ term in $ψ$ is presented, $M$ can move into the accepting state. We use one free atomic proposition to allow the choices. It can also wait since $v_2 \models b*$ and then upon receiving $v_5$ it will go to the accepting state.

Further non-determinism needs to be considered when two consecutive terms in $ψ$ use the $*$ operator. For example, the input traces $ρ_1 = \langle v_1, v_4 \rangle$, $ρ_2 = \langle v_1, v_2, v_4 \rangle$, $ρ_3 = \langle v_1, v_3, v_4 \rangle$, and $ρ_4 = \langle v_1, v_2, v_3, v_4 \rangle$, where $v_1(a) = v_2(b) = v_3(c) = v_4(d) = true$, all satisfy the formula $ψ = a; b*; c*; d$. $M$ needs to allow for enough choices on the states corresponding to the $*$ terms to accept the four traces. For $m$ consecutive $*$ operators, $x_i; x_{i+1}; \ldots; x_{i+m-1};*$, we consider the corresponding $m$ states $S^\ast = \{s_i, s_{i+1}, \ldots, s_{i+m-1}\}$ with all transitions possible from state $s_k \in S^\ast$ to state $s_p \in S^\ast$ where $k < p$ on the same input valuation. Therefore, we need $\lceil \log_2 m \rceil$ atomic propositions to encode these transitions as $(s_k, x_k \land choice(k, p, \bar{r}), s_p)$ where $\bar{r}$ is the vector of additional atomic propositions and $choice$ is a unique choice of a valuation of propositions in $r$ mapped to $p$ and $k$. The same applies to terms in $ψ$ that follow $s_i$ such that $y_i = \epsilon$ and $y_{i-1} = \ast$.

### A. Equisatisfiable DFA construction

Let $A' = A \cup \bar{r}$ where $\bar{r}$ is the vector of additional atomic propositions. $T'$ is the set of SERE terms where $A'$ is the set of atomic propositions, and $V'$ is the set of valuations where $A'$ is the set of atomic propositions.

We construct the transition function $δ$ by constructing four partial transition functions.

The function $δ_0$ denotes the transitions at the initial state. $δ_0 = \{(s_0, r), (s_0, r \land x_1, s_1)\}$.

The function $δ_1$ is the transitions corresponding to terms $x_i y_i \ i \in \{1 \ldots n\}$ where $y_i = \epsilon$ and $y_{i+1} \neq \epsilon$.

The function $δ_2$ is the transitions corresponding to terms $x_i y_i \ \ i \in \{1 \ldots n\}$ where $y_i = \ast$. $δ_2 = \{(s_1, s_{i+1}, s_{i+1}) \ | \ m \leq i \leq n \text{ and } s_i, s_{i+1} \in Q \text{ and } y_i = \ast \}

\cup \{(s_t, \neg x_{i+1}, s_0) \ | \ 0 \leq i < n \text{ and } s_i, s_0 \in Q \text{ and } y_i = \epsilon \}$

The function $δ_3$ is the transitions corresponding to terms $x_i y_i \ i \in \{1 \ldots n\}$ where $y_i = \ast$. $δ_3 = \{(s_i, \bigwedge_{j=i}^n \neg x_j, s_0) \ | \ m \leq i \leq n \text{ and } y_i = \ast \}

\cup \{(s_i, x_j \land t_{ij}, s_j) \ | \ i < j \leq m \leq n \text{ and } \forall k, i < k \leq m \implies y_k = \ast \text{ and } t_{ij} = \text{choice}(i, j, \bar{r}) \vee \forall k, i < k \leq m \implies \neg x_k \}$

The function $δ_4$ is the transitions corresponding to terms $x_i y_i \ i \in \{1 \ldots n\}$ where $y_i = \ast$. $δ_4 = \{(s_i, \bigwedge_{j=i}^n \neg x_j, s_0) \ | \ m \leq i \leq n \text{ and } y_i = \ast \}

\cup \{(s_i, x_j \land t_{ij}, s_j) \ | \ i < j \leq m \leq n \text{ and } \forall k, i < k \leq m \implies y_k = \ast \text{ and } t_{ij} = \text{choice}(i, j, \bar{r}) \vee \forall k, i < k \leq m \implies \neg x_k \}$

The difference between $δ_{4*}$ and $δ_{4*}$ is that in $δ_{4*}$ no self transitions are defined.

The transition function $δ$ is now defined as $δ = δ_0 \cup δ_1 \cup δ_2 \cup δ_3 \cup δ_{4*}$.

Finally, we construct $F$ the set of accepting states as follows. If $y_n = \epsilon$, then $F = \{s_n\}$. If $y_n = \ast$ then $F = \{s_i \ | \ i = n \or k < i \leq n \text{ and } \forall j, k < j < n \implies y_j = \ast \}$. Intuitively, this includes the states corresponding to the suffix of terms with $*$ including one preceding term.

For example, the accept states in $M$ corresponding to the formula $x_1; x_2; x_3; x_4$ are $F = \{s_2, s_3, s_4\}$.

**Theorem 1** (Equisatisfiability of $M$ and $ψ$). A formula $ψ = x_1 y_1; x_2 y_2; \ldots; x_n y_n$ where $A = \{x_1, x_2, \ldots, x_n\}$, and a constructed DFA $M = (Q, \epsilon, F, V', T', δ), M$ and
ψ are equisatisfiable. In addition if there exists ρ that satisfies M then ρ also satisfies ψ.

The proof is by induction on the length of the formula ψ and is available in the online appendix [2]. Note that it is shown in the proof that the satisfiability of ψ and M will be by the same trace, with some existential quantification over the added free (auxiliary) atomic propositions.

B. Input to ABC

The ABC solver accepts an And-Inverted-Graph (AIG) sequential circuit as input. An AIG is a sequential circuit restricted to only use AND and NOT logical gates. The translation from a DFA to an equivalent AIG circuit is straightforward. In short, we encode each state from the DFA by a unique valuation of the register variables, and construct the initial values and the next state functions of the registers according to δ. The AIG circuit will also have a unique output o who is true only when the values of the registers correspond to a state in F. Note that o will be then negated in order to perform bounded model checking.

For a formula of the form ψ = φ₁ ∧ φ₂, we construct C₁ and C₂ that correspond to φ₁ and φ₂, respectively, and we use a conjunction of the outputs of C₁ and C₂ to correspond to the satisfiability of ψ. Similarly, we use a disjunction for φ₁ ∨ φ₂.

V. IMPLEMENTATION

We implemented our technique and integrated it with the ABC [18] synthesis and verification framework. We used ANTLR [19] to provide users with a C like input language, augmented with constructs that support wire declarations, synchronization, and SERE specifications. Our tool supports scalar variables, boolean variables, arrays, and functions including recursion.

The tool generates an AIG circuit as discussed in Section IV. The added free atomic propositions are left as free primary input variables into the AIG circuit.

The goal of the verification procedure is to ensure that there exists at least one setting of the primary input variables that leads the AIG representing the SERE specification ψ from its initial state to one of its accept states. Let R be the set of all possible valuations of r; |R| ≤ m where m is the maximum number of consecutive * operators in ψ since the size of r is bounded by log₂(m).

Our goal is to prove that ∃vᵢ ∈ R such that ψ is satisfied. We encode the existential quantifier with a disjunction over all the valuations in R.

If the system under test violates ψ, ABC returns a counterexample and our tool provides a user friendly debugging interface to debug the system.

VI. EXPERIMENTAL RESULTS

We compare our implementation with NuSMV2 [6], a symbolic model checker used for the verification of system designs. NuSMV2 accepts Computational Tree Logic, Property Specification Language, and LTL as specification languages. We compare our implementation with the NuSMV2 model checker for LTL properties.

In several examples, such as the load balancer example, we succeeded to generate an AIG and find counterexamples in defect circuits where other techniques in NuSMV2 failed.

All computation times provided in the following are obtained on a machine with 2.20 Ghz Intel Core i7 processors running an x64-version of Ubuntu Linux. The allowed memory usage is up to 8 GB and we set a timeout of 1800 seconds. For our experiments, we used NuSMV v2.5.4.

A. LILY [3] and UNBEAST [4] Examples

We used LILY examples [3] and the UNBEAST load balancer example [4] as benchmarks for comparison.
We passed LTL formulae from the benchmarks to LILY and UNBEAST and generated implementation designs. Then we translated the resulting designs manually into the input language of our tool as well as into SMV, the language of NuSMV2. In the cases where LILY and UNBEAST were not able to generate designs, we manually wrote dummy designs in which defects surely exist. Note that in both cases, we manually translated the LTL properties into SERE.

We passed the implementation annotated with the SERE specification to both NuSMV and to our tool and compared the results based on the size of the resulting structure passed to the model checker, and on the computation time. NuSMV generates BDDs to perform reachability analysis. Our tool generates AIG circuits and passes them to ABC. We used the number of latches and AND gates in our synthesized AIG before and after applying optimizations versus the number of states in the generated DFA and the total number of BDD nodes from NuSMV2. We use the commands dump fsm and print usage to obtain such information from the NuSMV2 tool.

Table I shows a summary of the results obtained from performing formal verification of the realizable load balancing examples from UNBEAST [4] and examples from the LILY suites [3]. Designs labeled as load_* correspond to load balancing examples, while designs labeled as demo-v* correspond to examples from the LILY benchmarks corresponding to a traffic light system. Note that we restrict our attention to realizable LTL formulae.

We used the demos, version 3 and 19, from the LILY benchmarks for comparison. We were able to generate the circuits and verify them in both cases. We employed several circuit level synthesis techniques available ABC [18] and were able to significantly reduce the size of the problem. NuSMV2 was also able to verify both models efficiently.

For the load balancer examples, we verified 5 out of the examples that we tested and we found problems and fixed them in the others. We used LILY and UNBEAST to generate the models from the specifications, and then checked the generated models against their specifications. NuSMV2 was also able to verify the 5 examples but failed (timed out at 30 minutes) to synthesize the LTL formulae for other benchmarks such as load_30, load_75, load_76, load_76, load_77, load_78, and load_79. UNBEAST and LILY were not able to generate a model of the specifications as well. Notice that the load_79 benchmark is the largest design in the load balancer benchmarks with 9 clients and a fixed number of servers.

This is evidence of the high utility of our technique which enables model checking where other tools fail. We also note that in all of the cases, the size of the problem we send to the model checker was smaller than the size of the problem generated by NuSMV2.

### B. IBM Arbiter case Study

We also used our tool against the IBM generalized buffer [16]. The model consists of four senders that communicate with a generalized buffer in order to send data to two receivers. Each sender has its own data line while the receivers share a common data bus. The buffer also includes a first-in first-out queue. We translated the VHDL implementation provided from IBM and checked it against the defined specifications. We checked for two assertions on the design.

- Sender requests are always acknowledged, and
- Arbiter requests are always acknowledged.
TABLE II
SIZE OF THE PROBLEM AND DECISION OF OUR TOOL ON THE IBM GenBuf Arbiter

| Assertion | Latches | Ands | Latches | Ands | Verification | Time(s) |
|-----------|---------|------|---------|------|--------------|---------|
| (1)       | 800     | 3209 | 0       | 0    | Verified     | 0       |
| (2)       | 792     | 3139 | 38      | 141  | Found counter | 0.61    |

Note that since the original LTL assertions are of the form “is always acknowledged”, writing an SERE specification for the good traces would not be useful for bounded model checking since the specification would match if one request was acknowledged once. In order to overcome this limitation, we can use a bound on the number of requests and then check that all requests within this bound have been acknowledged.

We were able to efficiently verify the first assertion. However when verifying the second assertion our implementation detected a counter example, and after debugging and inspection we found that there is a defect in the assignment of the request acknowledgments in the provided VHDL implementation. Table II shows the size of the synthesized AIG circuit in terms of number of latches and number of And gates before and after optimizations, and the verification decision of our tool for both specifications.

Our tool and the experiments are all available online.

VII. RELATED WORK

Several techniques have been developed in the literature in order to synthesize LTL formulae, usually describing properties that hold over real-life hardware systems and designs. These synthesis techniques have different targets, some aim to generate complete and correct systems based on input specifications, while others are targeted at generating monitors to ensure correct functionality of systems through assertion checking. We differ than most of the literature in that we synthesize an equisatisfiable circuit to the formula that is good to be used for model checking purposes only.

NuSMV2 [6] is a symbolic model checking tool that employs both satisfiability (SAT) and BDD based model checking techniques. It processes an input describing the logical system design as a finite state machine, and a set of specifications expressed in LTL, Computational Tree Logic and Property Specification Language. Given a model \( M \) and a set of specifications \( P \), NuSMV2 first flattens \( M \) and \( P \) by resolving all module instantiations and creating modules and processes, thus generating one synchronous design. It then performs a boolean encoding step to eliminate all scalar variables, arithmetic and set operations and thus encode them as boolean functions.

In order to avoid the state space explosion problem, NuSMV2 performs a cone of influence reduction [14] step in order to eliminate non-needed parts of the flattened model and specifications. The cone of influence reduction abstraction technique aims at simplifying the model in hand by only referring to variables that are of interest to the verification procedure, i.e. variables that influence the specifications to check [11]. We use NuSMV2 to compare the results of our implementation on a set of benchmarks as described in Section VI.

FoCs is an industrial tool developed at IBM research labs, targeted at generating simulation checkers from formal specifications [5]. The tool’s goal is to reduce, or possibly eliminate the amount of human intervention in writing and maintaining functional checkers. FoCs takes input specification expressed in RCTL [20], and generates formal checkers written in VHDL. These checkers are then linked with the original VHDL and executed on a set of test programs. The role of the formal checkers is to make sure that the original design never goes into an error state.

The generation of the formal checkers from the RCTL specifications is done in three steps. First, the RCTL is translated into a NDFA according to the algorithm described in [20]. This NDFA will have a set of error states, which represent the states that the design should never go into if it meets the required specifications. In order to be able to generate the VHDL checkers, the NDFA has to be translated into a DFA, which is in turn translated into a VDHL process. This process will then be run alongside the original design to check for violations of the specifications.

The key drawback of FoCs’ approach is that transformation algorithm generates a DFA that can be exponential in the number of states of the NDFA, which takes us back to the state-space explosion problem. The authors claim that such a limitation does not exist in their case, since the simulation is rather sensitive to the number of VHDL lines in the generated checker, which is at most quadratic in the size of the property to check. Our approach differs from FoCs in that it aims at generating
a AIG free primary input variables that is linear in the size of the property, without generating an intermediary NDFA. Therefore, it can help rendering the generated VHDL checker even smaller in terms of the lines of code.

Jobstmann et. al developed LILY [3], a synthesis tool aimed at synthesizing correct designs from LTL specifications. It is implemented on top of Wring, and introduces several optimizations based on alternating tree automata, covering both game based and simulation based optimization techniques. They present an incremental algorithm for checking realizability of LTL formulae, and output a Verilog [21] model in case the formula is realizable. We made use of LILY to generate several design models, and then we checked these generated models against their original specifications using our own implementation.

UNBEAST [4] is a synthesis tool that aims to generate system designs that are correct by construction. It takes as input a specification containing environment assumptions and system guarantees, and splits them into safety and non-safety conditions. Each of these sets of conditions are then handled differently in the synthesis game. Unlike LILY, it relies on universal co-Büchi word automata instead of co-Büchi tree automata. It checks for realizability of LTL formulae and returns SMV models when realizable. We differ from both UNBEAST and LILY in the type and the goal of synthesis. Our goal is to generate monitor from SERE properties, while LILY and UNBEAST generate models that satisfy the LTL properties. Our generated DFA is equisatisfiable to the input SERE property, and thus can be used for model checking purposes only.

VIII. CONCLUSION

In this paper we presented a technique that takes a formula in SERE and transforms it into an AIG circuit with a number of states that is linear in terms of the length of the formula. The generated circuit is equisatisfiable to the formula and enables the use of symbolic model checking and bounded model checking where it was not possible before; i.e. where the typical translation from N DFA equivalents of the formula to a DFA blows up exponentially.

REFERENCES

[1] Accelera, “Property specification language reference manual,” 2004.
[2] M. Daniele, F. Giunchiglia, and M. Vardi, “Improved automata generation for linear temporal logic,” in Computer Aided Verification. Springer, 1999, pp. 681–681.
[3] B. Jobstmann and R. Bloem, “Optimizations for ltl synthesis,” in Formal Methods in Computer-Aided Design, 2006. FMCAD ’06. IEEE, 2006, pp. 117–124.
[4] R. Ehlers, “Symbolic bounded synthesis,” in Computer Aided Verification. Springer, 2010, pp. 365–379.
[5] Y. Abarbanel, I. Beer, L. Gluhovsky, S. Keidar, and Y. Wolfsthal, “Focus–automatic generation of simulation checkers from formal specifications,” in Computer Aided Verification. Springer, 2000, pp. 538–542.
[6] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, “NuSMV 2: An opensource tool for symbolic model checking,” in Computer Aided Verification. Springer, 2002, pp. 359–364.
[7] G. Holzmann, “The model checker spin,” Software Engineering, IEEE Transactions on, vol. 23, no. 5, pp. 279–295, 1997.
[8] T. Krof, Introduction to formal hardware verification. Springer, 1999.
[9] R. H. Hardin, Z. Har’El, and R. P. Kurshan, “Cospan,” in Computer Aided Verification. Springer, 1996, pp. 423–427.
[10] M. Y. Vardi, “Automata-theoretic model checking revisited,” in Verification, Model Checking, and Abstract Interpretation. Springer, 2007, pp. 137–150.
[11] E. M. Clarke, O. Grumberg, and D. A. Peled, Model checking. MIT press, 1999.
[12] R. Armoni, S. Egorov, R. Fraer, D. Korchemny, and M. Y. Vardi, “Efficient ltl compilation for sat-based model checking,” in Proceedings of the 2005 IEEE/ACM International conference on Computer-aided design. IEEE Computer Society, 2005, pp. 877–884.
[13] M. Sipser, Introduction to the Theory of Computation. Thomson Course Technology Boston, MA, 2006, vol. 27.
[14] S. Berezin, S. Campos, and E. M. Clarke, Compositional reasoning in model checking. Springer, 1998.
[15] R. K. Ranjan, A. Aziz, R. K. Brayton, B. Plessier, and C. Pixley, “Efficient bdd algorithms for fsm synthesis and verification,” IWLS’95, Lake Tahoe, CA, vol. 253, p. 254, 1995.
[16] S. Rabinovich, “Generalized buffer (psl/sugar version),” http://www.research.ibm.com/haifa/projects/verification/RB_Homepage/tutorial3/index.html, IBM Haifa Research Lab.
[17] S. Rodger and T. Finley, JFLAP: An Interactive Formal Languages and Automata Package. Jones & Bartlett Learning, 2006.
[18] R. Brayton and A. Mishchenko, “Abc: An academic industrial-strength verification tool,” in Computer Aided Verification. Springer, 2010, pp. 24–40.
[19] T. Parr and R. Quong, “Antlr: A predicated-ll (k) parser generator,” Software: Practice and Experience, vol. 25, no. 7, pp. 789–810, 1995.
[20] I. Beer, S. Ben-David, and A. Landver, “On-the-fly model checking of rctl formulas,” in Computer Aided Verification. Springer, 1998, pp. 184–194.
[21] D. Thomas and P. Moorby, The Verilog® Hardware Description Language. Springer, 2002, vol. 2.